USENIX The Second USENIX Symposium on Mobile and Location-Independent Computing *Ann Arbor, Michigan* *April 1995*

conference

proceedings

The Second USENIX Symposium on Mobile and Location-Independent Computing

Ann Arbor, Michigan
April 10–11, 1995

Sponsored by
The USENIX Association

USENIX® The UNIX® and Advanced Computing Systems Professional and Technical Association

ISBN 1-880446-69-3

Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.

USENIX Association

Proceedings of the Second USENIX Symposium on Mobile and Location-Independent Computing

April 10-11, 1995

Ann Arbor, Michigan, USA

Table of Contents

Second USENIX Mobile & Location-Independent Computing Symposium

Ann Arbor, Michigan
April 10-11, 1995

Program Committee

Readers

Preface

Fifty-eight authors from nine countries submitted twenty-two papers to the Second USENIX Mobile and Location-Independent Computing Symposium. Of those twenty-two papers, we were able to program twelve.

It was my privilege to serve as Program Chair, but much of the hard work was done by others. I would particularly like to thank Ellie Young, Judy DesHarnais, and the rest of the staff at the USENIX office for their hard work. Peter Honeyman offered many useful suggestions throughout the months of planning. Chris Tanis of CITI helped with the Program Committee mailings. Andy Adamson and Mary Simoni helped coordinate the musical events at the evening reception, and my father, Gerald Rees, suggested the Hands-On Museum and was instrumental in obtaining permission to hold the reception there.

I am grateful to the Program Committee and Readers for the many hours they put in. And finally, my thanks to all the authors, panelists, and speakers for contributing their time to the Symposium.

Jim Rees
Program Chair

A Cluster-based Approach for Routing in Ad-Hoc Networks*

P.Krishna M. Chatterjee N. H. Vaidya D. K. Pradhan
Department of Computer Science
Texas A&M University
College Station, TX 77843-3112
Contact E-mail: pkrishna@cs.tamu.edu

Abstract

This paper presents a "cluster-based" approach to routing in ad-hoc *networks. A* cluster *is defined by a subset of nodes which are 'reachable' to each other. Our approach is motivated by our study of existence of clusters (size greater than 2) in random graphs. The basic idea behind the protocol is to divide the graph into number of overlapping clusters. A change in the network topology corresponds to a change in the cluster membership. Performance of the proposed routing protocol (reconvergence time, and update overhead) will hence be determined by the average cluster size in the network graph. The effectiveness of this approach lies in the fact that existing routing protocols can be directly applied to the network – replacing the nodes by clusters. When the average cluster size is less than 2, the proposed approach does not perform any worse than the existing routing protocols. Generalization of the proposed approach is a subject of ongoing research.*

1 Introduction

Mobile wireless networks gives users communicating capability and information accessing capability regardless of the location of the user. With the availability of wireless interface cards, mobile hosts are no longer required to remain confined within the static network premises to get network access. In order to communicate with any particular host, it is first necessary to locate the host in the network. This is due to the fact that the hosts are mobile and could be anywhere. In addition to mobility, the host can also be in a disconnected mode (power-saving). This dynamic feature in mobile wireless networks leads to the problem of keeping track of the topology connectivity. This problem becomes noticeable when the the rate of change is high, and the network sizes are large.

An important issue in mobile wireless networks is the design and analysis of topology management schemes. *This paper investigates the consequence of mobility and disconnections of mobile hosts on the routing overhead in a "mobile" network.* We define a *mobile* network as a cooperative set of mobile hosts which can communicate with each other over the wireless links (direct or indirect) without any static network interaction[1]. Example of such networks are *ad-hoc* networks [1, 3, 21], and packet radio networks [2, 16, 17]. The term *ad-hoc* network is in conformance with current usage within the IEEE 802.11 subcommittee [21]. Ad-hoc networking in the wireless world refers to the ability to create a peer oriented network between several clients all of which are wireless and all of which are "virtually LAN'd" together. Ad-hoc also implies that the wireless network can be created dynamically or in an "ad-hoc" fashion. Once this type of network has been created by two clients then other users may freely gain media access (provided the specific security and configuration parameters of the physical link are valid). The wireless LANs and their standards address only the MAC and PHY layers and thus a wireless network which features this function relies on the upper level protocol stacks (i.e., IPX, IP, netBIOS, etc.) to allow for either peer-to-peer or client-server operation from a session/application point of view. The focus of this paper is to introduce a new routing methodology more suited for such *mobile* networks.

Example applications of such *mobile* networks range from conference rooms to battlefields. To communicate with each other, each mobile user needs to connect to a static network (wide area network, satellite network). However, there might be situations where connecting each mobile user to a static network may not be possible due to lack of facilities, or may be expensive. In such situations, it would be more preferable for the mobile users to set up communication links between themselves without any static network interaction [3].

In the current proposed mobile wireless networks, routing information of each mobile host is maintained in some database (*HLR* and *VLR* in *IS*-41 [5, 6], *home agent* and *foreign agent* in mobile *IP* [8, 9]) which is located in the static network. However, there is no such database available for *ad-hoc* networks. The routing information will be maintained at the mobile hosts to forward packets to other hosts. The problem in hand is the complexity of updating the routing information

*Research reported is supported in part by AFOSR under grant F49620-94-1-0276, and Texas Advanced Technology Program under grant 999903-029.

[1] We assume that a mobile host has the capability to communicate directly with another mobile host. It is also assumed that the mobile hosts have the capability to forward (relay) packets.

in such a dynamic (due to the mobility of the hosts, and limited power on the hosts, thus, host disconnections) network.

1.1 Previous Work

Numerous routing protocols have been proposed in the recent years. One of the most popular techniques for routing in communication networks is via distributed algorithms for finding shortest paths in weighted graphs [12, 13, 14, 18]. These distributed algorithms differ in the way the routing tables at each host are constructed, maintained and updated. The primary attributes for any routing protocol are :

- Simplicity : This is one of the most primary attributes for a routing protocol. Simple protocols are preferred for implementation in operational networks [1].
- Loop-free : At any moment, the paths implied from the routing tables of all hosts taken together should not have loops. Looping of data packets to be routed results in considerable overhead.
- Convergence characteristics : The time required to converge to new routes after a topology change should not be high. Quick convergence is possible by requiring the nodes to frequently broadcast the updates in the routing tables.
- Storage overhead : The memory overhead incurred due to the storage of the routing information should be low.

The conventional routing protocols can be broadly classified as *distance vector* and *link state* protocols. The *distance vector* routing uses the classical distributed bellman-ford algorithm [11, 16, 18, 19]. Each host maintains for each destination a set of distances through each of its neighbors. In order to maintain up-to-date information, each host periodically broadcasts to each one of its neighbors, its current estimate of the shortest path to every other host in the network. For each destination, the host determines a neighbor to be the next hop for a message destined for the destination if the neighbor has the shortest path to the destination.

Link state routing requires each host to have knowledge of the entire network topology [20]. To maintain consistent information, each host monitors the cost of each communication link to each of its neighbors, and periodically broadcasts an update in this information to all other hosts in the network. Based on this information of the cost of each link in the network, each host computes the shortest path to each possible destination host. The processing overhead and the network bandwidth overhead of *link state* protocols are generally more than *distance vector* protocols.

The problems in using conventional routing protocols in an ad-hoc network have been discussed in great detail in [1, 3]. For completeness sake, we briefly list the problems in the following.

- The conventional routing protocols were not designed for networks where the topological connectivity is subject to frequent, unpredictable change as evident in ad-hoc networks. Most of them exhibit their least desirable behavior for highly dynamic interconnection topology.
- Existing protocols could place heavy computational burden on mobile computers, and the wireless networks, in terms of battery power and network bandwidth respectively.
- Convergence characteristic of these protocols is not good enough to suit the needs of ad-hoc networks.
- Wireless media has a limited and variable range, different from existing wired media.

The protocol described in [1] addresses some of the above stated problems by modifying the Bellman-Ford routing algorithm. They use sequence numbers to prevent routing table loops, and, settling-time data for damping out fluctuations in route table updates. The convergence on the average was rapid, however, the worst case convergence was non-optimal. Moreover, their protocol required frequent broadcasts of the routing table by the mobile hosts. The overhead of the frequent broadcasts goes up as the population of mobile hosts increases.

A distributed routing protocol for mobile packet radio networks was proposed by Corson et al. [2]. Similar to [10], routing optimality was of secondary importance. Rather, their goal was to maintain connectivity between the hosts in a fast changing topology. Moreover, instead of maintaining distances from all sources to a destination, the protocol guarantees route maintenance only for those sources that actually desire routes. This property helps in reducing the topology update overhead. The protocol is a compromise between two extremes : *flooding* [11] (suited for high rate of topology change), and *shortest-path* algorithms [15] (suited for low rate of topology change).

1.2 Proposed Approach

This paper presents a new methodology for routing and topology information maintenance in mobile wireless network. Our approach is motivated by our study of existence of clusters (size greater than 2) in random graphs. The basic idea behind the protocol is to divide the graph into number of overlapping clusters. A change in the network topology corresponds to a change in the cluster membership. Performance of the proposed routing protocol (reconvergence time, and topology update overhead) will then be determined by the average cluster size. The effectiveness of this approach lies in the fact that existing routing protocols can be directly applied to the network – replacing the nodes by clusters. When the average cluster size is less than 2, the proposed approach does not perform any worse than the existing routing protocols. For future reference, let us formally define *clusters*.

Definition 1.1: A *k-cluster* is defined by a subset of nodes which are 'reachable' to each other by a path of length at most k for some fixed k. A *k-cluster* with $k = 1$ is a clique. This paper deals with clusters of $k = 1$, i.e., *1-clusters*. (Hereafter, we refer *1-cluster* simply as cluster.) However, we can also generalize our protocols with values of k greater than one (subject of our ongoing research). Each cluster is identified by its members.□

Definition 1.2: The *size*, $S(C)$ of a cluster C is the number of nodes in C.□

Definition 1.3: A graph is *cluster-connected* if it satisfies the following two conditions :

1) The union of the clusters cover the whole graph.
2) There is a path from each node to every other node through the edges of the clusters in the graph. □

The main problem here is to develop protocols for cluster maintenance. The protocols should be simple and distributed, and, should incur low overhead. To this effect, we develop simple distributed protocols to detect, and, build irredundant clusters in a graph. We maintain a *minimal* number of clusters based on the connectivity criteria (Definition 1.3). Experiments are performed to determine the average cluster size in random graphs. Section 2 presents the problem of routing in mobile wireless networks. We present the protocols to divide the nodes into clusters in section 3. Section 4 presents experimental results and its discussions. Section 5 presents the proposed routing protocol based on clusters. Conclusions are presented in section 6.

2 Preliminaries

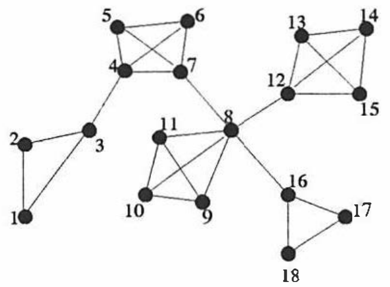
The problem addressed in this paper can be defined as follows:

Given: A wireless mobile network configuration.

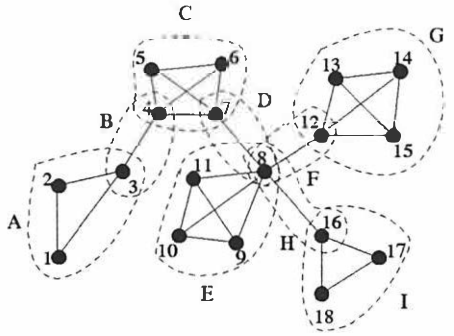
Problem: Find a 'good' loop-free routing between each mobile host in the network, where the topological connectivity is subject to frequent unpredictable change.

The problem requires a *loop-free distributed routing* protocol which determines an acyclic route between each host whenever a change in the topology is detected. The protocol is intended for use in networks where the rate of topological change is not so fast as to make "flooding"[2] the only viable routing method, but not so slow as to make any static topology routing applicable. A loop-free[3] routing is a routing where the path from one host to another does not traverse through the same node twice. A loop-free routing is desirable to minimize the consumption of resources during routing.

A 'good' route from one host to another is not necessarily the shortest path. In an environment of frequent topological change, a 'good' route connects a host to another host and the route length is comparable to the shortest one. Each host maintains a datastructure describing the network topology and some routing information pertaining to a common routing protocol. The routing protocol adapts asynchronously in a distributed fashion to arbitrary changes in topology in the absence of global topological knowledge. Let an undirected graph, $G = (V,E)$ represent the



(a)



(b)

Figure 1: An Example

network of mobile hosts. Each node u, in the graph denotes a mobile host H_u. Due to the limited range of wireless transreceivers, a mobile host can communicate with another host only within a limited geographical region around it. This region is called the host coverage area – d being the radius. The geographical area covered by a host coverage area is a function of the medium used for wireless communication. A host H_u is in the vicinity of H_v if the distance between nodes u and v is less than or equal to d. An edge (u,v) connects node u and node v if the corresponding hosts are in the 'vicinity' and have a *direct connection* between each other. A host may sometime be isolated where it has no other mobile hosts in its vicinity. Such a host will be represented in the graph by a disconnected node. A host H_{v1} is connected to another host H_{v2} if there exists at least one path from node $v1$ to $v2$. The path length is given by the number of edges on the path. Routing from one node to another node should ideally use the path with the shortest length. The wireless mobile routing problem requires a distributed graph algorithm to determine a loop-free route from each node to every other node.

Example 2.1: The graph (in Figure 1(a)) is formed based on the geographical locations of the 18 mobile hosts. In this example, the graph is connected as each

[2]Flooding can be described as an algorithm whereby a node broadcasts a message packet to its neighbors, who in turn broadcast the packet to all their neighbors, except the neighbor from which it was received. This process goes on till the message packet reaches the intended destination. This happens provided the destination is connected to the node which originated the flood [2].

[3]Loop-free routing requires prevention of loops in the routing tables. Here, existence of temporary loops are not of concern.

node is *reachable* to every other node. It can be observed that based on the positions, some nodes form clusters. The graph (in Figure 1(a)) can be divided into nine clusters (in Figure 1(b)). The clusters and their respective members are as follows : A (1,2,3), B (3,4), C (4,5,6,7), D (7,8), E (8,9,10,11), F (8,12), G (12,13,14,15), H (8,16) and I (16,17,18). Routing can be done from one node to another by only using clusters. Routing from node 1 to node 16 is done through the clusters A, B, C, D and F. The graph in Figure 1(b) is *cluster-connected* because, (i) the union of the clusters covers the whole graph, and (ii) there is a path from each node to every other node using the clusters. □

A topological change in the mobile host network corresponds to a change in the graph structure $G(V,E)$ to $G'(V',E')$. A change in the graph structure can be of the nature of a node (host) or an edge (connection between two hosts). We outline four types of events in the mobile network that can incur changes in the graph (in the following H_A and H_B are mobile hosts) :

A) H_A switching ON: A host H_A switching ON will include itself in the graph and make connection with all the hosts in its 'vicinity'. Hence, $V' = V \cup \{A\}$ and $E' = E \cup \{(u,A)$, s.t. H_u is connected to $H_A\}$.

B) H_A switching OFF: A host H_A switching OFF will exclude itself from the graph and delete all its edges. Hence, $V' = V - \{A\}$ and $E' = E - \{(u,A)$, s.t. $(u,A) \in E\}$.

C) H_A gets connected to H_B: Here, an edge between A and B will be added to the graph. Hence, $V' = V$ and $E' = E \cup \{(A,B)\}$

D) H_A gets disconnected from H_B: Here, the edge between A and B will be removed from the graph. Hence, $V' = V$ and $E' = E - \{(A,B)\}$

A routing protocol will change its routing information based on the afore-mentioned four types of changes in the graph. We add some definitions and properties which will assist in describing our proposed routing protocol.

Definition 2.1: Cluster set S_n of a node n is defined as the set of all clusters in which n is a member. □

Definition 2.2: If a cluster $C \in S_n$ can be removed and still all nodes $i \in C$, have paths to every other node j, where $j \neq i$ and $j \in C$, using other clusters in S_n, C is a *redundant* cluster. □

A cluster determined to be redundant for one node, may not be redundant for other nodes. A graph will have *irredundant* clusters if and only if each node n do not have *redundant* clusters in their cluster set S_n.

Definition 2.3: A node A is a *boundary* node if it is a member of more than one cluster. In Figure 1(b), node 3 is a boundary node as it belongs to two clusters, (1,2,3) and (3,4). However, node 1 is not a boundary node as it only belongs to (1,2,3).□

Lemma 2.1: Addition of each new node to the graph adds at least one new irredundant cluster[4]. [4] □

[4]However, one or more existing clusters can become redundant due to the addition of the new clusters.

Lemma 2.2: Given a graph with irredundant clusters, with the addition of new members, only the members of new clusters can identify the redundant clusters in the graph. [4] □

Lemma 2.3: Given a graph with irredundant clusters, removal of one node will reduce the count of the number of irredundant clusters by at most one. [4] □

3 Cluster Formation

Our proposed routing protocol is based on the formation of clusters. Hence, efficient cluster formation will be the crux of a routing protocol of this nature. Clusters should be formed in such a way that the resulting graph is *cluster-connected* (See Definition 1.3). Routing from one node to another will consist of routing inside a cluster and routing from cluster to cluster. A change in the mobile network may or may not result in a change in the cluster compositions. Here, we have assumed clusters with $k = 1$ (See Definition 1.1). As mentioned in Section 2, we have identified four different possible types of changes in the mobile network graph in the occurance of a single event.

We present an algorithm to divide the graph into clusters. Variations and optimizations of the algorithm are not ruled out [4]. The main contribution of this paper is to present the effectiveness of the cluster-based approach for routing in mobile networks. We will now present the protocols for cluster updates with each type of topological change.

3.1 Host H_A switches ON

The new graph structure $G'(V',E')$ is formed with the added node. The new node A will result in at least one new cluster so that with the cluster, node A can route to the rest of the graph. However, if A connects two disjoint subgraphs, it may result in more than one added cluster. These new clusters are denoted by *essential* clusters and can be detected by A itself. The addition of new clusters may result in zero or one or more clusters being redundant. The two tasks performed during the topological change are (i) addition of new clusters, and (ii) removal of redundant clusters. The goal is to have *minimal* number of clusters such that the graph remains *cluster-connected*. The protocol initiated by new node A is described as follows.

```
     Procedure Switch ON($A$);
     Begin;
1.   $A$ sends messages to its neighbors about its
     new arrival;
2.   For each of $A$'s neighbors $n$ do
3.       send list of its neighbors to $A$;
4.   $A$ gets information from all neighbors &
     creates all possible new clusters list;
5.   $A$ executes Find Essential(A,list);
6.   $A$ broadcasts Essential Clusters;
7.   For each of $A$'s neighbors $n$ s.t.
     $n \in$ Essential Clusters do
8.       $S_n = S_n \cup$ Essential Clusters ;
9.       $n$ executes Find Redundant ($n$, $S_n$) ;
10.      $n$ broadcasts Redundant Clusters;
11.  Change in cluster structures are propagated
     to rest of the graph;
     End;
```

The new node A broadcasts a message to its neighbors indicating its new arrival. Upon receipt of the new arrival message, the neighbors send a list of their neighbors to A. Based on the information received from its neighbors, A determines all possible clusters and stores them in *list*. The new node A then executes **Find Essential** function.

```
    Function Find Essential($A$, $list$);
    Begin;
1.  Sort the clusters in $list$ in a non-descending
    order of their sizes;
2.  For each cluster $C \in list$ do
3.      $Mark(C) := essential$;
4.  For each cluster $(C \in list) \wedge$
    $(Mark(C) = essential)$ do
5.      For each node $(n \in C) \wedge (n \neq A)$ do
6.          For each cluster $(C' \in list) \wedge$
            $(C' \neq C) \wedge (Mark(C') = essential)$ do
7.              if $(n \in C')$
8.                  $Mark(C) := non\text{-}essential$;
9.                  break;
10.     if $(Mark(C) = essential)$
11.         $Essential\ Clusters :=$
            $Essential\ Clusters \cup C$;
12. return;
    End;
```

The **Find Essential** function sorts the clusters in *list* in a non-descending order of their sizes. Initially all the clusters are marked *essential*. It then goes through each *essential* cluster C to find if a node (other than the new node A) in C is a member of any other *essential* clusters. If so, it marks the cluster C as *non-essential*. This will ensure that a node (other than the new node A) is a member of no more than one *essential* cluster. Moreover, since the clusters are sorted in a non-descending order of their sizes, the **Find Essential** function returns the largest clusters possible. The *essential* clusters determined by **Find Essential** function are stored in *Essential Clusters*.

```
    Function Find Redundant($n$, $S_n$);
    Begin;
1.  For each cluster $C \in S_n$ do
2.      $Mark(C) := redundant$;
3.      For each node $(i \in C) \wedge (i \neq n)$ do
4.          $match$ := FALSE ;
5.          For each cluster $(C' \in S_n) \wedge (C' \neq C)$ do
6.              if $(n \in C')$
7.                  $match$ := TRUE ;
8.          if $match$ = FALSE
9.              $Mark(C) := non\text{-}redundant$;
10.     if $(Mark(C) = redundant)$
11.         $S_n := S_n - C$ ;
12.         $Redundant\ Clusters :=$
            $Redundant\ Clusters \cup C$ ;
13. return ;
    End;
```

The new node A then broadcasts the *essential* cluster information its neighbors. Only the neighbors who are members of the *essential* clusters are involved in searching for redundant clusters. A neighbor will be a member of no more than one *essential* cluster. The neighbor then adds the *essential* cluster to its cluster set. Addition of the *essential* cluster might make one or more existing clusters in its cluster set *redundant*. The neighbor then executes **Find Redundant** function. This function determines *redundant* clusters based on *Definition 2.2*. The *redundant* clusters determined by **Find Redundant** function are stored in *Redundant Clusters*. The neighbor broadcasts the *redundant* cluster information to its cluster mates. The neighbor and its cluster mates then remove the *redundant* clusters.

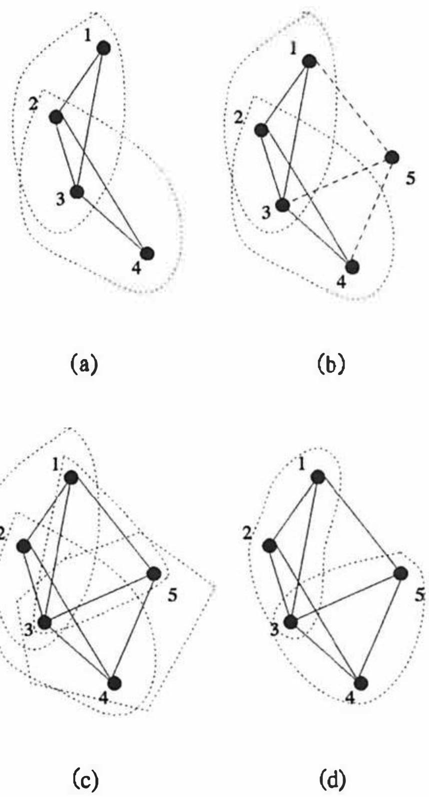


Figure 2: An Example of a Node Addition

Example 3.1: For an easier understanding, Figure 2 gives an example involving a graph with 4 nodes. Figure 2(a) has 4 nodes and two clusters, namely, (1,2,3) and (2,3,4). When node 5 is switched ON, it sends messages to nodes 1, 3, and 4 (Figure 2(b)). On receiving information back from the nodes 1, 3 and 4, node 5 forms clusters (1,3,5) and (3,4,5) as seen in Figure 2(c). It chooses (3,4,5) as the *essential cluster* and broadcast it to nodes 3 and 4. In the redundant removal phase, node 3 detects the cluster (2,3,4) to be redundant. The final clusters are (1,2,3) and (3,4,5) as in Figure 2(d).□

Please refer to [4] for the formal proofs.

3.2 Host H_A switches OFF

When host H_A turns OFF, its disappearance will only be detected by its neighbors. Hence, its neighbors will initiate a protocol to adapt to this topological change. We first illustrate the protocol with an example.

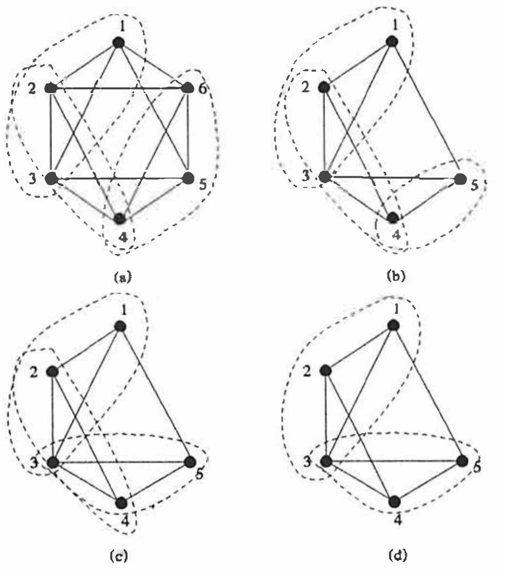


Figure 3: An Example of a Node Removal

Example 3.2: Figure 3 shows the cluster formations when a node is turned OFF in a graph. Figure 3(a) has six nodes with three clusters, namely, (1,2,3), (2,3,4) and (4,5,6). When node 6 is turned OFF, the cluster (4,5,6) shrinks to (4,5) (Figure 3(b)). In the next step, nodes 4 and 5 try to expand their cluster and creates a cluster (3,4,5) (Figure 3(c)). In the redundant removal phase, node 3 detects the cluster (2,3,4) to be redundant. The final clusters are (1,2,3) and (3,4,5) as in Figure 3(d). □

There could be more than one node detecting the removal of a node. The number of nodes initiating the **Switch OFF** procedure should be the number of clusters A was a member. A node i detecting the removal of node A will initiate the procedure if and only if no other member of the cluster in which node i and A are members has already initiated the procedure.

Let B be one of the nodes detecting host H_A turning OFF. The procedure initiated by node B is described as follows.

```
    Procedure Switch OFF(A,B);
    Begin;
1.  B requests the list of neighbors from
    the cluster mates of the shrinked cluster;
2.  For each cluster mate n do
3.      send list of its neighbors to B;
4.  B gets this information and tries to expand
    the shrunk cluster ;
5.  B broadcasts the new cluster ;
6.  For each cluster mate n do
7.      $S_n = S_n \cup$ new cluster;
8.      n executes Find Redundant (n, $S_n$) ;
9.      n broadcasts Redundant Clusters;
10. Change in cluster structures are propagated
    to rest of the graph;
    End;
```

We outline two possible cases:

1. Node A was not a *boundary* node: A will be only contained in a single cluster. Hence the cluster shrinks after the removal of node A. The other members of the cluster try to expand the cluster size by including nodes from a different cluster. This might result in redundant clusters. The redundant clusters are detected and removed.

2. Node A was a boundary node: Here, the removal of node A will cause more than one cluster to shrink in size. This will give an opportunity to the shrinked clusters to combine and in that process create redundant clusters. All nodes n that were neighbors to A will look for clusters which can use itself, neighbors of A, and nodes from a different cluster. With the formation of new clusters, the redundant clusters are detected and removed.

The new cluster structures are then sent out to the rest of the graph.

3.3 Host H_A gets connected to Host H_B

The new connection between H_A and H_B is detected simultaneously by both the nodes. Both of them now try to create a cluster involving A, B and nodes from other clusters. Formation of these clusters will be consistent with A as well as B as the cluster includes both A and B. Once the cluster is formed, the other cluster-mates of A and B look for redundant clusters if any.

3.4 Host H_A disconnects Host H_B

Here, we identify two cases as follows.

1. A was not in any cluster with B: The topological change will result in no change in any of the clusters.

2. A and B shared common clusters: Here, the topological change will result in the shrinking of the involved clusters. The hosts H_A and H_B initiate the **Switch OFF** protocol.

At the event of the change of a cluster structure, the neighbors will propagate the change to the rest of the graph.

4 Experimental Results

We performed experiments to determine the average size of clusters in random graphs. The clusters were determined using the **Switch ON** procedure described in Section 3.1.

4.1 Experimental Framework

Input to the simulations are (i) N (number of nodes), and (ii) ρ (ratio of the total area to the host coverage area (D/d)). We assume a square room to be the total area, where, D is the dimension of the square room. The value of d was chosen to be 10 units (The typical range of infra-red transreceivers which is commonly used for indoor wireless communication is of the order of 10 meters). The node positions ((x, y) coordinates) were chosen randomly using uniform distribution within the total area i.e., (0,0) to $(D-1, D-1)$.

4.2 Results and Discussions

For each N (10, 20, 30, 40, 50), ρ (2, 5, 10, 15, 20), we ran the simulations for 10000 iterations. We believe that the results obtained are transient-free. It should be noted that *maximal* clusters are always sought. It was noticed that for a connected graph, the sequence of **Switch ON** procedures always produced a *cluster connected* graph. As shown in Figure 4, the average cluster size increases as N increases. It also increases when ρ decreases. For example, the average cluster size will be very high in a small crowded room. Any cluster size bigger than 2 will benefit the protocol. Figure 4 shows the region of values of N and ρ where the average cluster size is greater than 2. In these scenarios, clustering will benefit.

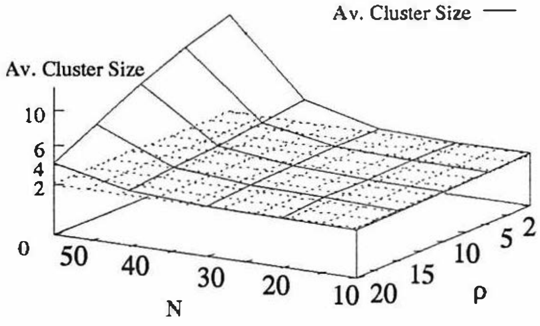


Figure 4: Average Cluster Size Vs. N and ρ

As ρ grows (as it would for public wide area services), N must be large to get an average cluster size of 2 or more. However, the effectiveness of the proposed approach is that the existing routing protocols can be directly applied to the network – replacing the nodes by clusters. Thus, when the average cluster size is less than 2, the proposed approach does not perform any worse than the existing routing protocols.

5 Routing Protocol

Through an example, we will show in this section how to extend existing routing protocols to support cluster-based approach. We will extend a standard distance vector routing protocol [11, 16, 18, 19] to support clusters.

We will first discuss the necessary data structures to be maintained at each node for the routing protocol. We will then present an overview of the extensions to the standard protocol. Later in the section we will compare the performance of the proposed cluster-based approach with couple of existing routing protocols.

5.1 Data Structures

The following tables are maintained at each node :

- *ClusterTable* : This table provides the mapping between the nodes and their clusters. It might so happen that a node is a member of more than one cluster. Thus, for each node, the identifiers of all the clusters in which the node is currently a member, is maintained.

- *RouteTable* : For each destination cluster, the node maintains the identifier[5] of the next hop node, say n, and the number of hops it will take to reach a node in the destination cluster through n. This is the table which is referred to while routing a packet. This table maintains the shortest available path to every destination cluster.

- *AllRouteTable* : For each destination, this table maintains route information of all possible paths from the node. This table is used to determine the shortest available path to each destination node, which is maintained in *RouteTable*.

The *RouteTable* and *ClusterTable* for network in Figure 1 are shown in Tables 1 and 2. The *AllRouteTable*, for Figure 1, happens to be same as its *RouteTable* (Table 1), because, there is just one possible path between any two clusters.

ClusterId	*NextHop*	*Hops*
A	4	2
B	4	1
C	-	0
D	7	1
E	7	2
F	7	2
G	7	3
H	7	2
I	7	3

Table 1: *RouteTable* at node 6

5.2 Protocol

A routing protocol can be divided into two phases, namely, *route construction* and *route maintenance*. During the *route construction* phase, routes are constructed between all pairs of nodes. The *route maintenance* phase takes care of maintaining loop-free routes in the face of unpredictable topological changes.

[5]In *ad-hoc* networks, MAC address can be used to transmit packets directly to that node [3].

Node	*ClusterIds*
1	A
2	A
3	A, B
4	B, C
5	C
6	C
7	C, D
8	D, E, F, H
9	E
10	E
11	E
12	F, G
13	G
14	G
15	G
16	H, I
17	I
18	I

Table 2: *ClusterTable* at node 6

5.2.1 Route Construction Phase

The protocol to divide the network graph into clusters have been explained earlier. After clustering, each *boundary* node forwards the cluster information (i.e., cluster id and its members) to the other clusters it is part of. Along with the cluster information, a hop counter is included. The hop counter keeps track of the number of hops needed to reach any *boundary* node of that cluster. The *boundary* node of the new cluster increments the hop counter to 1 before forwarding the cluster information. If a *boundary* node gets information of a new cluster, it stores the cluster information in its *ClusterTable*, and the hop information in *AllRouteTable* and *RouteTable*. It increments the hop counter and then forwards the cluster information. A *boundary* node has to forward information of a new cluster only once.

Let us illustrate it with an example. In Figure 1, the *boundary* nodes are 3, 4, 7, 8, 12, and 16. Node 3 will send {A, (1, 2, 3)} (hop counter for A=1) to node 4 in cluster B, and, send {B, (4)} (hop counter for B=1) to nodes (1, 2) in cluster A. Since, 4 receives information of a new cluster (A), it increments the hop counter for A to 2 and forwards the cluster information of A. Thus, the *boundary* node 7 will get this information, increment the hop counter and forward the cluster information too. In this manner, the information of all clusters are distributed to all the nodes.

Upon receipt of information of all the clusters, the data structures *RouteTable*, *ClusterTable*, and *AllRouteTable* at each node will have the topology information of the whole network.

Each message packet will contain the identifier of the destination node in its header. When a node receives a message packet, it looks up the *ClusterTable* to determine the cluster in which the destination node is currently a member. Using the identifier of the destination node's cluster it looks up the *RouteTable* to determine the next hop node for the packet's destination. The node then forwards the message packet to the next hop node. This process of forwarding continues till the packet reaches its destination.

5.2.2 Route Maintenance Phase

This phase begins when there is a change in the network topology (host connection/disconnection, link failure/recovery). The route maintenance in our approach basically boils down to cluster maintenance. The protocols for cluster maintenance have been explained previously. The new cluster information will be propagated throughout the network (optimizations are not ruled out – they are not yet investigated). Loop freedom can be achieved using techniques suggested in the existing literature [1, 2] e.g., sequence numbers, link status, etc.

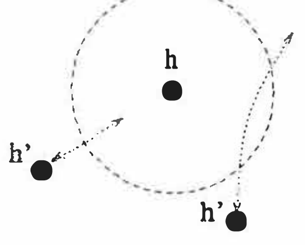


Figure 5: Movements That Cause Unnecessary Link Creations/Deletions

5.3 Implementation Details

- Detection of a new link : Each host periodically broadcasts a beacon, which, includes its identifier. If a host h receives a beacon from another host h' which is not in its current neighbor set, it means that there is a prospective new link to be created. However, the **Switch ON(h')** procedure is not immediately initiated. This is to avoid unnecessary oscillations due to the host h' moving in and out of host h's vicinity. Figure 5 shows the scenarios where the movement of h' could cause a sequence of unnecessary link creations/deletions.

- Detection of a link break : If a host h does not receive a periodic beacon from h' which is one of its cluster mates, it will assume that either h' has moved out of its vicinity (cluster) or that h' is disconnected. Host h will then follow the procedure for host disappearance as explained in Section 3.2.

5.4 Performance Evaluation

5.4.1 Comparison with *DSDV* [1]

- Our approach does not require the frequent broadcasts of routing tables to the neighbors as long as there is no change in a cluster membership. The proposed approach will however incur some form of cluster maintenance overhead as explained in Section 5.3. However, the size of a periodic beacon from each host is much smaller than the size of a routing table.

- Quick reconvergence in *DSDV* is obtained by quick re-broadcast by each and every recipient of the broadcast, causing degradation of the availability of the wireless medium. However, in our approach, re-broadcast is done **only** by the *boundary* nodes. Nodes other than *boundary* nodes just listen.

- Memory overhead due to storage of data structures are considerably smaller for the cluster-based approach when the average cluster size is more than 2. This is due to the fact that the routing information in our protocol is cluster-based which is smaller than node-based in *DSDV*.

5.5 Comparison with Corson's Protocol [2]

- Routing optimality is of secondary importance in [2]; finding a route is what mattered. This reduces the topology update overhead, because, as long as there is a route to a destination available, any changes in link status to that destination will not cause new routes to be searched and created. If the goal of our approach were to be similar to [2], our approach will incur lower topology overhead because of the fact that broadcasts and rebroadcasts are done **only** by the *boundary* nodes.

- The novel property of the protocol in [2] is that the routing is "source-initiated". Instead of maintaining distances from all sources to a destination, the protocol guarantees route maintenance only for those sources that actually desire routes. This property helps in reducing the topology update overhead. Our approach does not restrict a routing protocol to maintain routes between all pairs of nodes. However, if we were to maintain routes to a destination for only those sources that actually needs them, the performance of our approach will be "at least" same as [2].

6 Conclusion

Proposed in this paper is a new methodology for routing in mobile wireless networks. This paper shows that routing protocols based on clusters could obtain performance improvements over previous approaches. Cluster-based protocols allow the network to enjoy the liberty of maintaining routes between all pairs of nodes at all times, without causing much network overhead. Thus, a compromise on routing optimality as suggested in [2] to avoid network congestion might not be required.

Similar to [2, 10] the cluster-based approach does not guarantee shortest path. This is due to the fact that *maximal* clusters are always sought in the proposed approach. We are currently involved in the analysis of the routing overhead. Routing overhead is ratio of the path length between the source and the destination as determined by the proposed algorithm and the actual shortest path length between them. We expect the path length determined by the cluster-based approach to be comparable to the shortest path length. The tradeoff of a routing algorithm in such high-rate topological change environment such as ad-hoc networks and packet radio networks, is between the overhead due to topology update messages, and the routing overhead. We are also currently analyzing the overhead due to cluster maintenance and the topology updates.

This paper dealt with protocols for discrete events (host connection/disconnection, etc.). Future work will involve extensions of these protocols to support concurrent events. This paper assumes that the transmitted packets are received correctly, i.e., reliable links. We are currently investigating schemes to tolerate corrupt wireless links.

Apart from providing connectivity in a dynamic topology, maintaining routing information (in mobile wireless networks) has other advantages such as, extending the base station area coverage (Figure 6); consequence of that is delayed and might be much smoother handoffs.

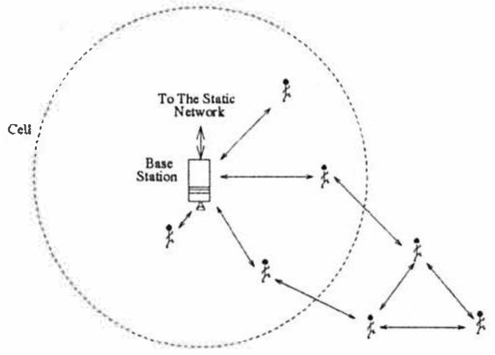


Figure 6: Extending base station coverage area

7 Acknowledgments

We thank the reviewers for their many valuable comments.

References

[1] C. Perkins, P. Bhagwat, "Highly Dynamic Destination Sequenced Distance Vector Routing (DSDV) for Mobile Computers," *Proc. ACM SIGCOMM Symposium on Communication, Architectures and Protocols*, pp. 234-244, 1994.

[2] M. Scott Corson, A. Ephremides, "A Distributed Routing Algorithm for Mobile Wireless Networks," to appear in *ACM Journal on Wireless Networks*, Vol.1, No.1, 1995.

[3] David B. Johnson, "Routing in Ad Hoc Networks of Mobile Hosts," *Proc. of Workshop on Mobile Computing and Applications*, Dec., 1994.

[4] P. Krishna et. al., "Routing in Mobile Networks," Technical Report (in preparation), Dept. of Computer Science, Texas A&M University.

[5] S. Mohan and R. Jain, "Two User Location Strategies for Personal Communication Services," *IEEE Personal Communications*, Vol. 1, No. 1, 1994.

[6] P. Krishna, N. H. Vaidya and D. K. Pradhan, "Efficient Location Management in Mobile Wireless Networks," Technical Report, Dept. of Computer Science, Texas A&M University, Feb., 1995.

[7] P. Krishna, N. H. Vaidya and D. K. Pradhan, "Location Management in Distributed Mobile Environments," *Proc. of Third Intl. Conf. on Parallel and Distributed Information Systems*, pp. 81-88, Sep., 1994.

[8] J. Ioannidis, et. al., "IP-based Protocols for Mobile Internetworking," *Proc. ACM SIGCOMM Symposium on Communication, Architectures and Protocols*, pp. 235-245, October 1991.

[9] C. Perkins, "IP Mobility Support," Internet Draft, IETF Mobile IP Working Group, Oct., 1994.

[10] E. Gafni, D. Bertsekas, "Distributed Algorithms for Generating Loop-free Routes in Networks with Frequently Changing Topology," *IEEE Trans. on Comm.*, January, 1981.

[11] D. Bertsekas, R. Gallager, *Data Networks*, Prentice-Hall, 1987.

[12] J. J. Garcia-Luna-Aceves, "A Unified Approach to Loop-free Routing Algorithm Using Distance Vectors or Link States," *ACM SIGCOMM Symposium on Communication, Architectures and Protocols*, Sep., 1989.

[13] J. M. Jaffe and F. M. Moss, "A Responsive Routing Algorithm for Computer Networks," *IEEE Trans. on Communications*, pp. 1758-1762, July, 1982.

[14] M. Schwartz and T. E. Stern, "Routing Techniques used in Communication Networks," *IEEE Trans. on Communications*, pp. 539-552, April, 1980.

[15] J. J. Garcia-Luna-Aceves, "Loop-Free Routing Using Diffusing Computations," *IEEE Trans. on Networking*, Vol. 1, No. 1, Feb., 1993.

[16] J. Jubin and J. D. Tornow, "The DARPA Packet Radio Network Protocols," *Proceedings of the IEEE*, Vol.75, No. 1, pp. 21-32, January, 1987.

[17] P. R. Karn, H. E. Price, and R. J. Diersing, "Packet Radio in the Amateur Service," *IEEE Journal on Selected Areas in Communications*, Vol. 3, No. 3, pp. 431-439, May, 1985.

[18] J. M. McQuillan and D. C. Walden, "The ARPA Network Design Decisions," *Computer Networks*, Vol.1, No.5, pp. 243-289, Aug., 1977.

[19] C. Hedrick, Routing Information Protocol, RFC 1058, June, 1988.

[20] J. M. McQuillan, I. Richer, and E. C. Rosen, "The New Routing Algorithm for ARPANET," *IEEE Trans. on Comm.*, Vol. 28, No. 5, pp. 711-719, May, 1980.

[21] W. Diepstraten, G. Ennis, and P. Bellanger, "DFWMAC - Distributed Foundation Wireless Medium Access Control," *IEEE Document P802.11-93/190*, Nov., 1993.

Handoff and System Support for Indirect TCP/IP

Ajay Bakre and B.R. Badrinath
Department of Computer Science
Rutgers University, Piscataway, NJ 08855.
Email: {bakre, badri}@cs.rutgers.edu

Abstract

Over the past few years, Transmission Control Protocol (TCP) has become the most widely used transport layer protocol on the Internet. TCP performs poorly however, if one of the communicating hosts is a mobile wireless computer [6]. One way to address this performance problem is to modify TCP to make it aware of host mobility. Such an approach is infeasible because of the sheer number of hosts on the Internet using TCP. Another way is to build a separate transport layer protocol for mobile hosts but this raises interoperability problems with the existing stationary hosts.

An indirect transport layer protocol such as I-TCP [5] allows using a separate transport protocol for the wireless link between a mobile host and its Mobile Support Router (MSR) without causing interoperability problems with the fixed network. Experiments with I-TCP have shown significant improvement in throughput over regular TCP [5]. Use of indirect protocols requires additional functionality at the MSRs. In this paper we describe the implementation of such functionality for I-TCP. We also present measurements for the time needed for handoffs in case a mobile host with an active connection switches cells.

1 Motivation

Transmission Control Protocol (TCP) [21] has become the most popular transport layer protocol on the Internet. Evolution of TCP has kept pace with the advances in communication technology and current TCP implementations perform fairly well over wide ranging data links from low speed serial links to high speed LANs. Congestion control and flow control mechanisms of TCP are widely accepted as standards in the Internet community and have largely contributed to the growth of Internet on which millions of host end systems depend for communication.

Experiments with TCP have demonstrated poor performance however, if one of the communicating hosts happens to be a mobile wireless computer [6]. Host mobility causes temporary disruption in the network layer connectivity resulting in loss of TCP segments. Error prone wireless links used by mobile hosts to communicate with the fixed network also contribute to the increased loss of TCP segments sent to or from mobile hosts. This loss of TCP segments triggers congestion control at the transmitting host which seriously limits the end to end throughput. The congestion control mechanism used by TCP is clearly too conservative when faced with host mobility and wireless links.

One way to address the problem mentioned above is to modify the TCP specifications to take host mobility into account so that congestion control steps in only in case of a genuine network congestion. There are two problems with such an approach. First, it is infeasible to modify all the existing TCP implementations because of the sheer number of hosts on the Internet using TCP. Second, it is extremely difficult to determine the exact cause of a packet loss at the two end points of a TCP connection which may span multiple hops over the Internet. As an example, genuine congestion conditions may prevail on the fixed network when a mobile host switches cells. Attributing any packet loss in such a case purely to mobility can worsen the congestion on the fixed network.

Another alternative is to build a separate transport layer protocol for mobile hosts with specialized congestion and flow control mechanisms. Such a protocol may rely on information from the lower layers and from the base station or the mobile support router (MSR) to distinguish between packet loss due to congestion and that due to mobility or wireless errors. Using a separate protocol however, raises interoperability problems with the existing stationary hosts.

An *indirect* transport layer protocol such as I-TCP [5] allows the use of a separate protocol stack for mobile hosts without causing interoperability problems with the fixed network. I-TCP is built on top of Columbia Mobile IP [10] and provides TCP compatible access to the fixed network from the mobile hosts *indirectly* through the MSR. A *mobility and wireless aware* transport protocol is used for communication between the mobile host and its MSR while regular TCP

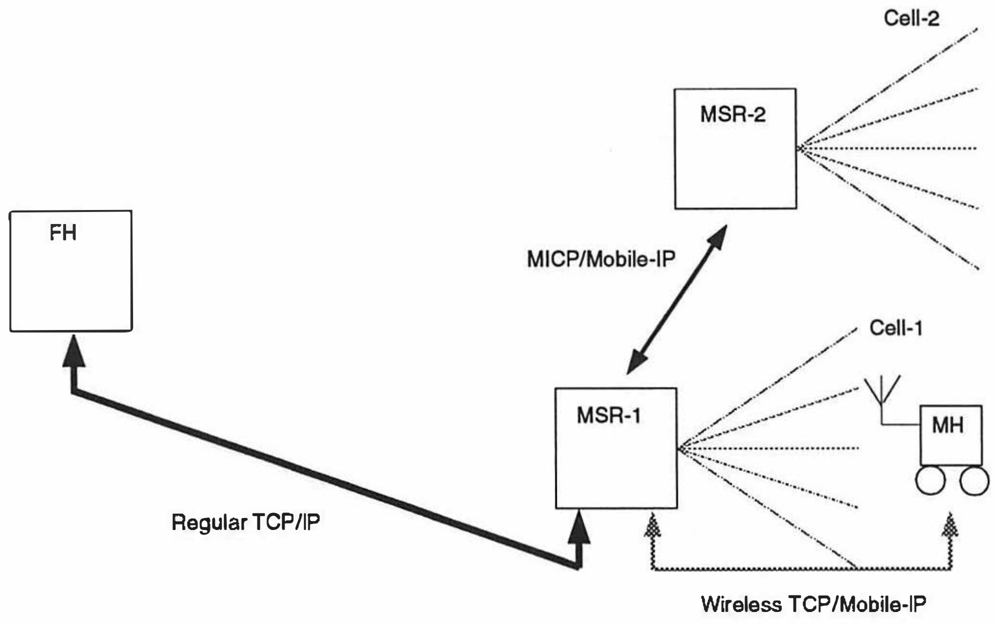


Figure 1. I-TCP connection between MH and FH via MSR-1

is used for communication between the MSR and the fixed hosts. No changes are thus needed to the TCP implementations on the fixed network. Experiments with I-TCP in the presence of host motion have shown improvements ranging from **1.5 to 4 times** in the end to end throughput compared to (regular) TCP [5].

Use of indirect protocols [4] requires specialized support for indirect connections and handoffs at the mobile hosts and the MSRs. In this paper we focus on the implementation of various mechanisms that we developed for the mobile hosts and the MSRs to support I-TCP connections and efficient handoffs in case a mobile host communicating over an open connection switches cells. Our scheme is not tied specifically with TCP and can be easily extended to support indirection for other transport layer protocols such as UDP [20] and RDP [27]. We present performance measurements for the time needed to achieve I-TCP handoffs when a mobile host moves from one wireless cell to another. We also analyze how the handoff time depends on the size of state information to be exchanged between two MSRs participating in the handoff.

The rest of this paper is organized as follows. The next section gives a brief overview of I-TCP. Section 3 describes the implementation of kernel and user level mechanisms needed to support I-TCP connections and handoffs. Section 4 deals with mobility management and the handoff procedure. Section 5 presents measurements on handoff performance. In section 6 we describe some applications that we modified to use I-TCP. Section 7 mentions some of the related research and section 8 contains concluding remarks.

2 Indirect TCP/IP Overview

We provide a brief overview of the I-TCP protocol in this section. More details on I-TCP can be found in [5]. As shown in figure 1, the central idea behind I-TCP is to break an MH to FH (end-to-end) TCP connection into two parts — one for the wireless link and the other for the wired network, the MSR serving the MH being the point where the break occurs. We can then easily modify the transport protocol on the wireless part (either TCP or some other lightweight protocol) without sacrificing interoperability with the TCP/IP hosts on the wired network. The following three features of I-TCP implemented over a suitable mobile IP scheme help to keep the indirection hidden from the fixed host (FH) communicating with the MH:

1. The MSR currently serving the MH (MSR-1 in the figure) *fakes* an image of the MH to the FH by using the IP address and port number of the MH for the wired part of the connection.
2. In case the MH switches cells, routing in mobile IP starts forwarding IP packets addressed to the MH to its new location i.e. the new MSR.
3. When an MH moves from one cell to another (e.g. from MSR-1 to MSR-2), the state of all its active I-TCP connections is handed over to the new MSR (MSR-2), which effectively restarts the connection

in the new cell *without* any participation from the communicating end points.

2.1 Establishing I-TCP connections

When a mobile host (MH) requests for an I-TCP connection to be established with a fixed host (FH), the MSR under which the MH is currently registered performs the following steps. It first establishes a regular TCP connection with the FH named in the connection request on behalf of the MH *using the IP address and port number of the MH for local endpoint parameters* - this constitutes the wired part of the I-TCP connection. Subsequently another connection is established between the MSR and the MH using a transport protocol that is tailored for mobile hosts and wireless links - this forms the wireless part of the I-TCP connection. In our implementation of I-TCP, we use TCP itself for the wireless part of the connection with one important modification *which resets the retransmission timer for the wireless part after a handoff causing an immediate slow start.* For the wireless side TCP connection, the MSR uses its own IP address and port number to identify the local endpoint. Thus an I-TCP connection between the MH and the FH consists of two separate TCP connections; and is uniquely identified by the following 3–tuple: {*<mh-address, mh-portnumber>, <fh-address, fh-portnumber>, <msr-address, msr-portnumber>*}.

2.2 Handing off I-TCP connections

When the MH switches cells (e.g. it moves from MSR-1 to MSR-2), the state associated with the two sockets at MSR-1 belonging to the I-TCP connection, is handed over to MSR-2. MSR-2 recreates the two sockets corresponding to the I-TCP connection with the *same* endpoint parameters that the sockets at MSR-1 had associated with them. Since the connection endpoints for both wireless and fixed parts of the I-TCP connection do not change after a move, there is no need to *reestablish* the connection at the new MSR. This also ensures that the indirection in the transport layer connection is completely hidden from the FH. The port numbers used by the MSR for the I-TCP sockets on behalf of an MH cannot conflict with the same port numbers used by the MSR for its own TCP connections or by other MHs using I-TCP connections in the same cell because the IP address is different for each host.

2.3 End-to-end semantics

One consequence of using I-TCP instead of TCP is that the acknowledgments are not end-to-end but instead we have separate acknowledgments for the wireless and the wired parts of the connection. Many applications that use TCP such as ftp however, also have some kind of support built-in for application layer acknowledgment and error recovery. Such acknowledgments are necessary because TCP acknowledgments only indicate that the data was received by the TCP layer at the receiving host and not necessarily by the receiving application. For such applications, use of I-TCP does not give rise to *weaker* end-to-end semantics in comparison to regular TCP, *assuming that there are no MSR failures.* MSR failure can cause loss of state information resulting in loss of indirect connections. These drawbacks may be a small price to pay however, considering the performance benefits of using indirect protocols [5].

In particular, I-TCP is ideal for throughput intensive applications such as ftp and Mosaic which provide built-in error recovery mechanisms to retry a failed data transfer. Such applications are likely to become popular as mobile clients begin to access information services from the fixed network. Interactive applications can also benefit from using I-TCP if the MH makes frequent moves, in which case a TCP connection can back off into long pauses [6] but an I-TCP connection would attempt a *slow start* immediately after every handoff.

3 I-TCP Implementation

In this section we describe the implementation of I-TCP components at the mobile hosts and at the MSRs that provide the additional functionality needed to support indirect connections and handoffs. We have implemented I-TCP on 386/486 PC-ATs running Mach 3.0 micro kernel and Unix server (MK84/UX40). All the mobile hosts and the MSRs in our laboratory have similar hardware configuration and they all use 2 Mbps NCR WaveLAN technology for wireless communication. All the MSRs are also attached to a wired 10 Mbps Ethernet. The MSRs run a version of UX server modified for Columbia's Mobile IP and I-TCP. The mobile hosts run the regular UX server derived from 4.3 BSD Tahoe release. User lever mhmicp and msrmicp processes modified for I-TCP run on the mobile hosts and the MSRs respectively and execute the Mobile Internetworking Control Protocol (MICP) [10] which takes care of beaconing and registration of mobile hosts in a wireless cell. In addition, a user level I-TCP daemon running at each MSR manages I-TCP connections originating from all the MHs within its cell. The I-TCP daemon also participates in handoffs with other MSRs when an MH switches cells.

3.1 I-TCP interface at the MH

To establish an I-TCP connection instead of a regular TCP connection from an MH to a fixed host, we provide special I-TCP library calls which are similar to the socket interface provided in Unix 4.3 BSD. These library calls must be used by the MH applications instead of the regular socket system calls (*connect, listen, accept* and *close*) for opening and closing an I-TCP connection. To send or receive data on such an I-TCP connection however, the MH can use the regular send and receive primitives. The I-TCP calls only provide a wrapper around the regular socket system calls to perform the necessary handshake with the MSR, the parameters of these library calls being the same as the corresponding socket system calls. A TCP-based application thus needs minimal changes to run on an MH with I-TCP. On the other hand, *no modification is needed*[1] *to the applications running on a fixed host to communicate with an MH using I-TCP.*

The I-TCP library calls also inform the local mhmicp process about active I-TCP connections. This information is used in the registration protocol executed by an MH entering a new cell to assist in I-TCP handoffs. The details of I-TCP calls are given below:

1. **itcp_init** — Initializes the I-TCP library data structures which maintain information on open I-TCP connections on a per process basis.
2. **itcp_listen** — Similar to the *listen* system call except that an indirect listening socket is also created at the current MSR by its I-TCP daemon on behalf of the MH with the same address and port number which identify the listening socket at the MH. Any connection attempt made after this call by a remote host will be intercepted by the indirect socket at the MSR which will cause the I-TCP daemon to connect to the MH listening socket thus completing the two parts of the I-TCP connection. The local mhmicp process is also informed about the creation of a listening socket at the MSR which records the corresponding port number information.
3. **itcp_accept** — Similar to the *accept* system call but the connection request is received from the current MSR in response to a connection attempt made by some remote host which was intercepted by the listening socket at the MSR. The wrapper around the *accept* system call provided by the I-TCP library however, makes the indirection transparent to the calling process by returning the peer address and port number of the remote host that initiated the connection (and not of the MSR). The local mhmicp process is also informed about the newly established I-TCP connection which records the endpoint parameters of the connection.
4. **itcp_connect** — Similar to the *connect* system call except that the connection request is sent to a special thread at the I-TCP daemon which in turn makes the connection attempt to the remote host address specified in the call. The call returns successfully only after the remote connection attempt has succeeded which means that on a successful return from *itcp_connect* both parts of the I-TCP connection are established. The local mhmicp process is informed about the new I-TCP connection.
5. **itcp_close** — Similar to the *close* system call for a socket. We need to close the connection at both the remote host and the MSR I-TCP daemon. We also need to inform the mhmicp process to delete the connection parameters from its tables.
6. **itcp_rioctl** — This I-TCP call is expected to provide some functionality of a socket *ioctl* call to control the indirect (remote) socket at the MSR. In addition, this call can be used by a higher layer to send control messages to the indirect socket which are interpreted by a corresponding higher layer at the MSR. One possible use of this might be to install a filter in the data stream in either or both directions e.g. put data compression and decompression filters at the MSR so that the data transmitted on the wireless link is always compressed while the data transmitted on the fixed link is not. This particular I-TCP function has not yet been fully implemented.

The I-TCP socket calls were implemented as a library rather than using a kernel based mechanism for the following reasons. First, we wanted minimal changes to be made in MH applications to switch from TCP to I-TCP and therefore the I-TCP calls export the same interface as the regular socket calls. Second, we wanted to allow the use of TCP and I-TCP connections in the same application depending on the kind of traffic expected on those connections. One example of such a *hybrid* application is described in section 6. Third, since we used a version of TCP itself as the transport protocol over the wireless link, we did not have to provide a separate socket family or a separate protocol number for I-TCP. Such a scheme would be appropriate however, in a mobile wireless environment where a different (possibly lightweight) transport protocol is used over the wireless link. Lastly, a library implementation is much easier to develop and debug than a kernel based mechanism.

3.2 MSR I-TCP daemon

The I-TCP daemon running on every MSR is responsible for managing I-TCP connections through that MSR for all the MHs that are currently local to the MSR. The daemon maintains two open sockets for each

[1] Other than a mechanism to recover from MSR failures, if necessary.

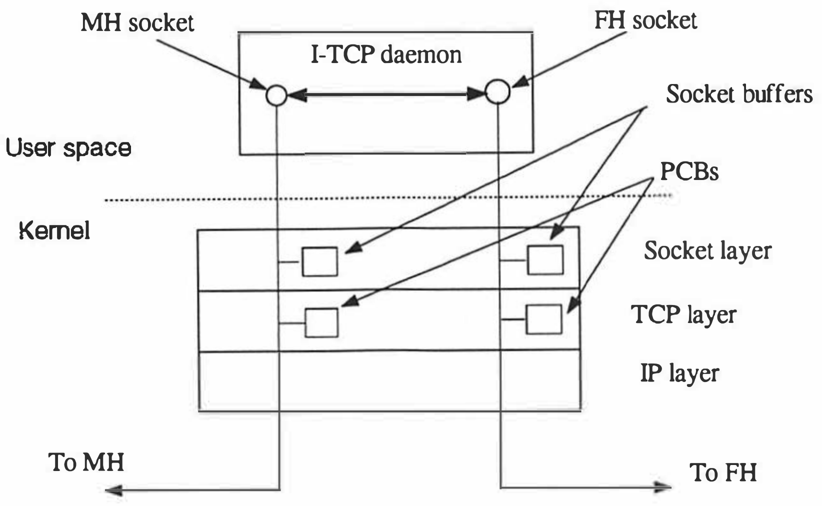


Figure 2. I-TCP connection state at the MSR

active I-TCP connection between a locally registered mobile host (MH) and a non-MSR fixed host (FH) as shown in figure 2. One of these sockets is used for communicating with the FH on the wired network and the other for communicating with the MH on the wireless link. The I-TCP daemon binds its FH side socket to the address and port number of the MH using an extended *bind* system call. Such a binding allows the MSR to fake an image of the MH to the FH which is unaware of the indirection. Binding to the MH address and port number also provides a mechanism at the IP layer in the MSR kernel to grab the TCP segments sent by the FH which are destined for the MH address and port number.

The I-TCP daemon is a threaded process [7] with different modules to communicate with the local MHs, the msrmicp process on the MSR and with the I-TCP daemons on other MSRs. These modules are described below:

1. **MH service module** — This module services I-TCP requests from the MHs local to the MSR. Such requests include those to open an I-TCP connection (either active or passive), to close a connection and remote *ioctls* on the indirect sockets. Servicing an I-TCP request from the MHs involves handshake with the requesting process and notification of success or failure of the requested operation. This module also starts up appropriate worker threads to operate a fully established connection between an MH and a fixed host (FH) or a listening thread for a listening indirect socket established in the name of an MH.
2. **MICP module** — This module handles the communication with the msrmicp process on the same machine (MSR). The msrmicp process executes the MICP protocol, which is part of the Mobile-IP software, to communicate with the local MHs and with the msrmicp processes at other MSRs. The I-TCP daemon depends on the msrmicp process for the information on the MHs entering and leaving the MSR's cell. In particular, an MH's entry in the cell causes the MICP module to establish all the indirect sockets that were active for this MH at its previous MSR. The information about all such connections is maintained by the MH itself and is passed to the msrmicp process of the new MSR as part of the registration protocol. The exit of an MH from the MSR's cell causes the initiation of handoff by the handoff module as described below.
3. **Handoff module** — This module manages cell handoffs for MHs entering and leaving the MSR's cell in cooperation with the handoff modules of the I-TCP daemons at other MSRs. A designated thread in this module listens for handoff requests from other MSRs. Handoffs are always initiated by the I-TCP daemon of the old MSR after a forward pointer is received by its msrmicp process from the new MSR. The I-TCP daemon at the new MSR must have already established the skeletons for the indirect sockets on behalf of the migrating MH for the handoff to be successful. On receipt of a handoff request from the old MSR, a handoff thread receives the state of each indirect socket and populates the previously established socket skeleton. After installing the state of each I-TCP connection, the handoff thread starts up worker threads (or a listening thread if needed), for the connection. The details of a cell handoff are described in a later section.
4. **Work module** — This consists simply of the worker

threads that copy the TCP data from the MH side of the connection to the fixed side and vice versa. Every I-TCP connection has two worker threads assigned to it - one for each direction. The MH worker thread receives data from the MH and sends it over to the fixed host, the FH worker thread does the opposite. The MH worker thread is also responsible for tearing down the connection and freeing any resources after it is shutdown by either the MH or by the FH. Both the worker threads check a handoff flag which is set by the MICP module when it discovers that an MH that was local has moved to another cell. The state of the two sockets (one for the MH side and the other for the fixed side) of the I-TCP connection can be safely handed over to the new MSR after the worker threads exit on finding the handoff flag set. It is possible to install a filter in either direction of the fully duplex stream of the I-TCP connection which is applied to the data sent in that direction. Such a filter can be installed during the connection setup time or even after that using a remote *ioctl* from the MH process. Such a filter may even send the data up to a higher layer for further processing before it is sent out on the outgoing link.

Managing the I-TCP connections at the MSR from a process in user space involves additional copying overhead on each half of a duplex connection. At the MSR data received from the wireless side (MH) on an I-TCP connection has to go up through the TCP and socket layers in the kernel and to the user space to the I-TCP daemon; and down again on the fixed side of the connection through the socket and TCP layers of the kernel to the IP output routine. On the other hand, a TCP/IP packet received from the MH over a direct connection would be forwarded by the IP layer in the kernel to the fixed network with nominal processing overhead.

We chose to build a user level daemon to manage I-TCP connections for two reasons. First, it is easier to build an indirect higher layer protocol such as RPC with such a scheme. Second, a user level daemon is easier to modify and debug than a full kernel implementation. The data copying overhead can be reduced by providing a mechanism in Unix that allows connecting two sockets in a way that the data received on one of them is directly copied to the send buffers of the other by the kernel. The end points of the two parts of the connections could still be maintained by the MSR I-TCP daemon thus retaining the flexibility afforded by a user level implementation[2]. The additional data copying overhead is not so critical for throughput intensive applications which are the primary focus of our implementation. This overhead will cause increased latency as seen by MH applications, however.

3.3 MSR kernel support

In the following subsections, we describe the modifications needed in the kernel networking code for I-TCP connections and handoffs.

3.3.1 IP layer support At any MSR, we allow binding sockets to the addresses and port numbers of MHs that are currently local to the MSR. This is essential to grab, on a per connection basis, the TCP packets which originate from fixed hosts and are addressed to a local MH. It also provides a mechanism using which the MSR can act as a *proxy* for the local MHs. The *in_pcbbind* function in the kernel verifies from the *mobinfo* data structure maintained by mobile IP, that the address named in such a bind request belongs to a locally registered MH. It is possible to modify the TCP code on the wireless side so that the connection between the MH and the MSR is *reconfigured* for the IP address and the port number at the new MSR after a move. Currently however, we keep the connection endpoints on the wireless side fixed even after a move for the sake of simplicity. For this purpose, at any MSR we also allow binding sockets to addresses and port numbers of other MSRs. The validity of an MSR address can again be checked by consulting the *msrinfo* structure maintained by mobile IP.

A small change is needed to the IP input routine at the MSR which should send the IP packets that are addressed to MH address and I-TCP port numbers up to the TCP layer at the MSR instead of forwarding them to the MH using mobile IP routing. A list of such I-TCP port numbers (which correspond to active connections) is maintained on a per MH basis by the MSR along with the *mhinfo* entry of the MH used for routing in mobile IP. The search cost to decide whether a TCP segment addressed to a local MH should be forwarded by the IP layer at the MSR or sent up to the TCP layer is thus proportional only to the number of I-TCP ports registered on behalf of the MH. We did not allocate a range of port numbers for I-TCP as it would have reduced the available port numbers for TCP and would have made *wild card* port allocation and matching more difficult.

3.3.2 Primitives for I-TCP handoff If a mobile host with an open I-TCP connection switches cells, handoff in mobile IP causes rerouting of IP packets addressed

[2] This is somewhat similar to DVMRP multicast routing with mrouted where the user level daemon dictates the multicast forwarding policy while the actual forwarding is done by the kernel.

Ioctl	Parameters	Description
SIOCCREATE	target socket, connection 4-tuple	Establish the connection endpoints of the target socket as if *bind* and *connect* have been called on the socket but without any communication with the peer host.
SIOCDELETE	target socket	Delete the target socket without any communication with the peer host.
SIOCGETSTATE	target socket, *struct socreq*	Capture the state of the socket listed in structure *socreq* and place it in the send buffer of the target socket.
SIOCSETSTATE	target socket, *struct socreq*	Retrieve and establish the state of the socket listed in the structure *socreq* from the receive buffer of the target socket.

Table 1. New socket *ioctls*

to the MH to the new MSR. To maintain the I-TCP connection therefore, the I-TCP daemon at the new MSR must reconstruct the two sockets for the wired and the wireless parts of the connection from the state information received from the I-TCP daemon at the previous MSR. This movement of socket should be completely transparent to the fixed host at the other end of the wired side connection. The socket state handoff needs to be fast enough so as not to become a performance bottleneck even with frequent moves. In addition, the data segments which are in transit at the time of handoff must be buffered at the new MSR because they cannot be processed until the full state information is available. This is necessary to avoid congestion on both sides (wired and wireless) of the I-TCP connection. Complete details of handoff are described in a later section. Here we restrict ourselves to describing the primitives needed to achieve a socket handoff.

We provided special *ioctl* calls in the MSR kernels to perform efficient socket handoffs. Two of these *ioctls*, namely SIOCGETSTATE and SIOCSETSTATE are used by the I-TCP daemons to reliably transfer the state information corresponding to a connected socket from the kernel of one MSR to that of another with minimum copying overhead. We also provided two other *ioctls* called SIOCCREATE and SIOCDELETE to create a *connected* skeleton socket which is waiting for its state to be installed and to silently delete a socket whose state has been transferred to another MSR. Table 1 summarizes the four *ioctls*. The *ioctl* mechanism was preferred over system calls primarily because it is much easier to add an *ioctl* than it is to add a system call to Unix. SIOCCREATE and SIOCDELETE operate on the I-TCP sockets whereas SIOCGETSTATE and SIOCSETSTATE operate on a handoff socket which is used for MSR-to-MSR state transfer. The latter two take *struct socreq* as an argument which lists the I-TCP socket descriptor whose state is to be sent or received. For the following discussion we will assume that the state of an I-TCP socket is being handed over from MSR-1 to MSR-2.

1. SIOCCREATE — This *ioctl* is used to establish the connection 4–tuple for a newly created I-TCP socket at MSR-2. The connection 4–tuple consists of local and foreign addresses and port numbers for the TCP connection. After this *ioctl*, the socket is marked as *moving* which means that it can accept incoming TCP segments and buffer them but these segments cannot be processed until the complete socket state becomes available at MSR-2.
2. SIOCDELETE — This *ioctl* is used to delete an I-TCP socket at MSR-1 after its state has been successfully handed over to MSR-2. It differs from the *close* system call in that it only deletes the socket data structures and does not attempt to close the connection with the remote peer.
3. SIOCGETSTATE — MSR-1 makes this *ioctl* call on a handoff socket with an established connection with MSR-2 to capture and send the state of an I-TCP socket after making sure that MSR-2 is prepared to accept the state of the I-TCP socket. The socket descriptor of the I-TCP socket is passed in the *struct socreq* argument to the *ioctl*. The TCP portion of this *ioctl* packs the TCP control block and any pending segments in the reassembly queue for the I-TCP socket. After this call, the I-TCP socket is marked as *frozen* which means it cannot send or receive any more data and it stays in this state until it is deleted by a SIOCDELETE call.
4. SIOCSETSTATE — MSR-2 makes this *ioctl* call on a handoff socket to receive and install the state of an I-TCP socket which is sent by MSR-1 with a corresponding SIOCGETSTATE call. The I-TCP socket must have been prepared by SIOCCREATE and marked *moving* prior to making this call, and

its descriptor is passed in the *struct socreq* argument to the *ioctl*. As part of this *ioctl*, the TCP control block of the I-TCP socket is re-initialized from the transferred state and the reassembly queue is also initialized. Also, if there are any pending TCP segments buffered for this connection which were received since the SIOCCREATE call, these segments are processed by calling the TCP input routine. The *moving* flag is cleared and the socket is ready for communication in both directions.

The SIOCGETSTATE *ioctl* avoids unnecessary copying of socket buffers and other state between kernel and user spaces by packing the send/receive buffers and state of the I-TCP socket inside the kernel and directly queuing it in the send buffer of the handoff socket. Similarly, SIOCSETSTATE avoids copying by retrieving the state and send/receive buffers of the I-TCP socket directly from the receive buffer of the handoff socket.

In addition to the above mentioned *ioctl*s, some minor changes were needed to the timer handling routines in TCP to inhibit retransmit timer handling on a socket which is marked as *frozen* or *moving*. Also, the TCP input routine needs to save segments received on a socket marked as *moving* in the reassembly queue for deferred processing. Any segments received on a *frozen* socket are silently dropped.[3]

4 Mobility Management — Interaction with Mobile IP

In Columbia mobile IP protocol a user level msrmicp daemon at the MSR is responsible for sending periodic beacons and for registration and expiration of mobile hosts. Similarly a user level mhmicp daemon running at each mobile host is responsible for listening to MSR beacons, sending greeting messages and informing the MSR about its previous location so that the MSR can send a forwarding pointer to the old MSR which was earlier routing IP packets for the MH. We integrated I-TCP handoff with the handoff in mobile IP since both need the information about MHs entering a cell and their earlier locations.

4.1 MICP i-entries

We extended the MICP protocol to carry information about I-TCP connections in the initial greeting that an MH entering a cell sends to the MSR managing the cell. At every MH, the mhmicp daemon maintains the endpoint information for each active I-TCP connection at the MH. Such information is maintained in MICP *i-entries* and it consists of the unique 3-tuple identifying the connection and some flags. The *i-entries* are updated from the messages sent by the I-TCP library linked with an MH application, when the application makes an I-TCP call. Unix domain sockets are used for communication between I-TCP library and mhmicp daemon. When an MH enters a new cell, i.e. when mhmicp daemon hears a beacon in the new cell after losing contact with the previous MSR, a greeting message (called MICP_GREET packet in mobile IP) is sent to the

Figure 3. Handoff algorithms for MSRs

New MSR:

```
wait for handoff request from old MSR
receive the MH address
lookup MH entry in the tables
for every I-TCP connection do {
  receive i-entry for connection
  lookup the connection entry
  receive and install state of
      MH side socket
  restart the wireless part of
      connection
  receive and install state of
      FH side socket
  restart the wired part of
      connection
  receive pending user level
      data buffers
  install MH and FH side user
      level buffers
  fork worker threads connecting the
      two parts of I-TCP connection
}
```

Old MSR:

```
locate the entry for MH that moved out
for every I-TCP connection do {
  signal worker threads to terminate
}
make a handoff connection to the new MSR
send MH address
for every I-TCP connection do {
  send i-entry for the connection
  freeze and send state of
      MH side socket
  freeze and send state of
      FH side socket
  send MH and FH side user level
      data buffers
  delete MH and FH side sockets
  free resources and delete MH entry
}
```

[3] In normal circumstances, no TCP segments should be received on a socket which is *frozen* because routing in mobile IP will carry all such packets at the new MSR (MSR-2).

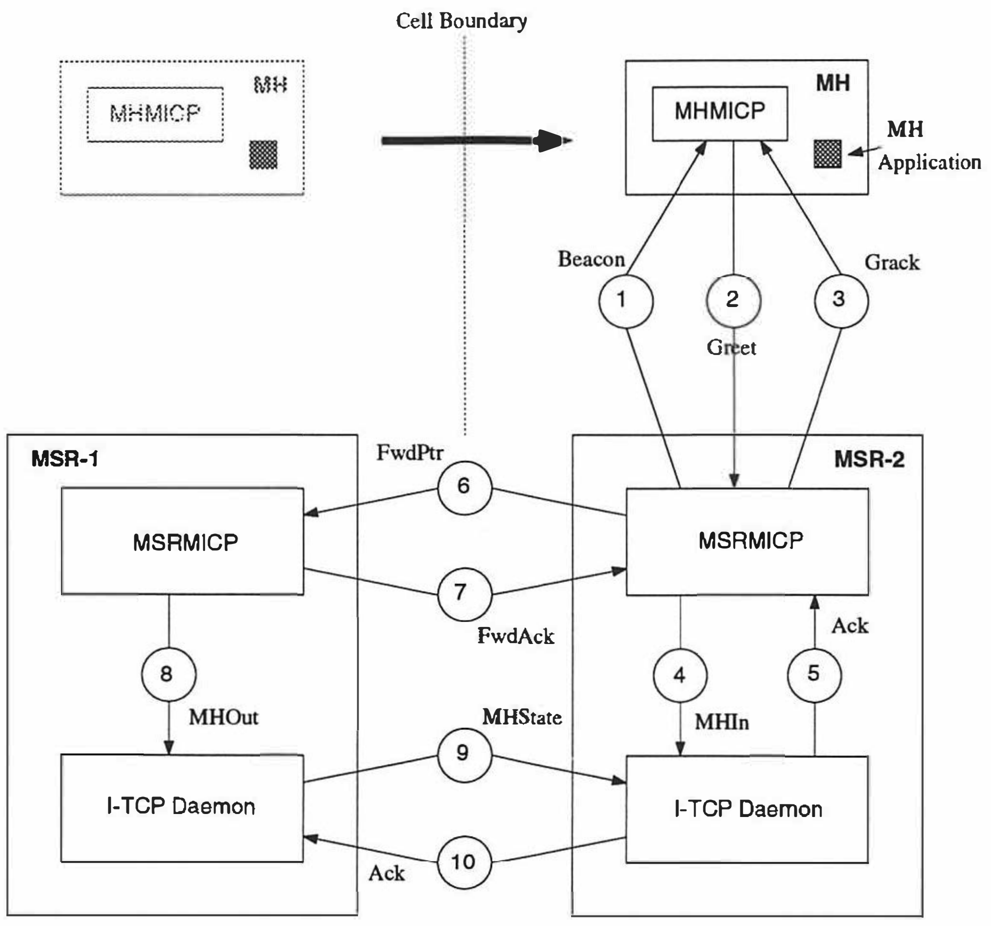


Figure 4. I-TCP / mobile-IP handoff sequence

new MSR. The mhmicp daemon sends its list of MICP *i-entries* in the initial greeting message in addition to the list of previously visited MSRs that may still believe that the MH is local to them.

4.2 Handoff sequence at MSRs

On the MSR side, when the msrmicp daemon receives an initial greeting from a new MH entering its cell which contains MICP *i-entries*, it sends a copy of the greeting message to the local I-TCP daemon. The I-TCP daemon establishes skeleton sockets for each I-TCP connection from the *i-entries* using SIOCCREATE calls. It also sends an ACK to the msrmicp daemon after which a forwarding pointer (MICP_FWDPTR) is sent to the previous MSR named in the greeting message.

When the msrmicp daemon at an MSR, where the MH had established I-TCP connections earlier, receives a forwarding pointer from another MSR, it updates its data structures to reflect the new location of the MH. It also sends a copy of the forwarding pointer to its local I-TCP daemon which then transfers the state of each I-TCP connection to the I-TCP daemon at the new MSR using a reliable protocol (TCP) and SIOCGETSTATE calls. The handoff algorithms executed by the new and the old MSRs are listed in figure 3. The I-TCP handoff sequence is shown in figure 4 and figure 5 shows the intermediate stages of an I-TCP connection during handoff.

5 Handoff Performance

We present measurements of handoff time for an I-TCP connection between a mobile host (MH) and a fixed host (FH) using different socket buffer sizes. We also analyze how the handoff time depends on the size of the state information to be transferred from one MSR to another. Fast handoffs are essential for the throughput improvement that I-TCP affords over regular TCP. Comparative throughput figures for I-TCP and TCP under a variety of wireless cell configurations can

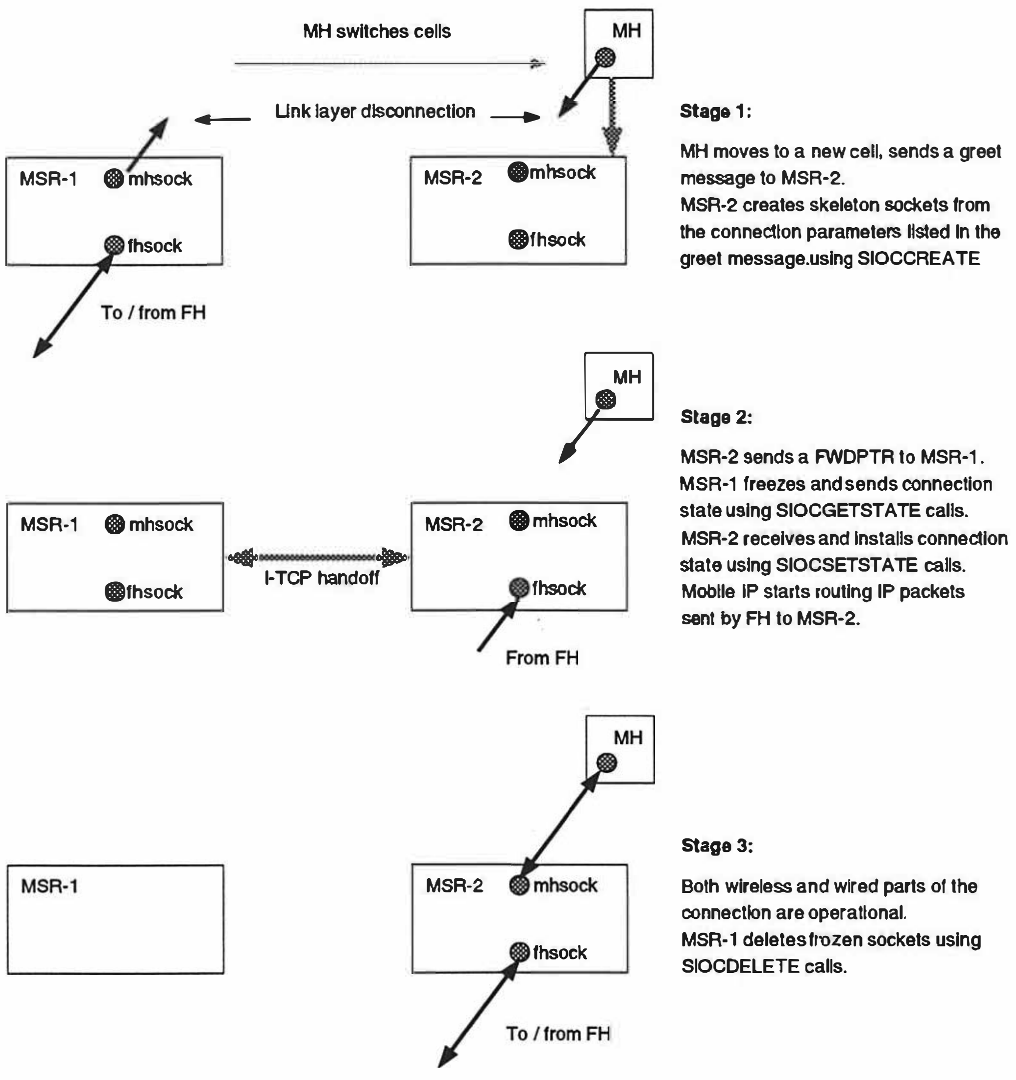

MH switches cells
MH
Link layer disconnection
MSR-1
mhsock
fhsock
To / from FH
MSR-2
mhsock
fhsock
Stage 1:
MH moves to a new cell, sends a greet message to MSR-2.
MSR-2 creates skeleton sockets from the connection parameters listed in the greet message.using SIOCCREATE
MH
Stage 2:
MSR-2 sends a FWDPTR to MSR-1.
MSR-1 freezes and sends connection state using SIOCGETSTATE calls.
MSR-2 receives and installs connection state using SIOCSETSTATE calls.
Mobile IP starts routing IP packets sent by FH to MSR-2.
MSR-1
mhsock
fhsock
I-TCP handoff
MSR-2
mhsock
fhsock
From FH
MH
Stage 3:
Both wireless and wired parts of the connection are operational.
MSR-1 deletes frozen sockets using SIOCDELETE calls.
MSR-1
MSR-2
mhsock
fhsock
To / from FH

Figure 5. Intermediate stages in I-TCP connection handoff

be found in [5][4]. The focus of this paper is on the handoff and system support we developed for I-TCP and therefore we will restrict ourselves to analyzing handoff performance.

We used the implementation platform described in section 3 for our handoff measurements. We measured the handoff time for an MH that had one I-TCP connection established with a fixed host 2 Ethernet hops away from the MSRs. The MH kept switching back and forth between two wireless cells managed by two MSRs at a constant rate (once every 10 seconds). The two wireless cells used in our experiments in fact overlapped, but we simulated non-overlapped cells using different MAC layer network IDs for the WaveLAN radio in the two cells. Cell switching was accomplished by a user process periodically changing the MAC layer network ID on the WaveLAN card at the MH which causes a loss of link layer connectivity with the current MSR and the MH starts listening to the new MSR. This user process also sends a signal to the mhmicp process immediately after changing the network ID, which forces mhmicp to look for a beacon from the new MSR. The beacon period used by the MSRs was 1 second which was the disconnection interval for the MH in the worst case. The handoff time was measured at the new MSR as the difference between the time when a greeting message arrives from the MH entering the cell and the time when I-TCP handoff completes.

Figure 6 shows the time taken by I-TCP connection handoff using different socket buffer sizes in a non-overlapped cell configuration where cell boundaries are

[4] Also available as Rutgers DCS technical report DCS-TR-314 (ftp://paul.rutgers.edu/pub/badri/itcp-tr314.ps.Z).

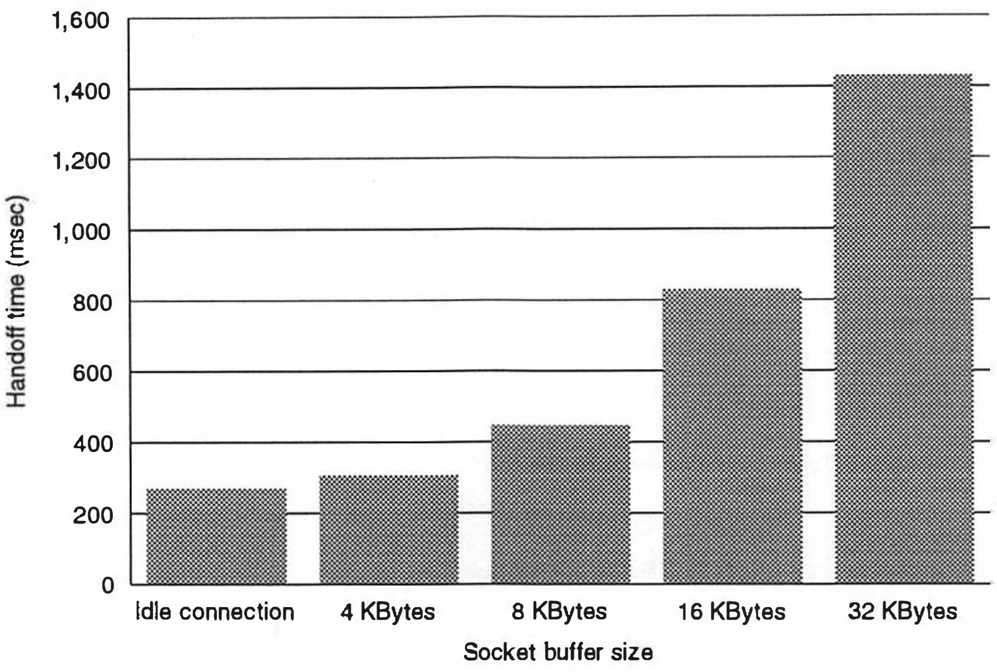


Figure 6. Handoff times for different socket buffer sizes

sharply defined and cell switching is instantaneous. The case marked *idle connection* denotes the handoff time for a connection on which no data is being sent. The graph shows that it takes about 265 milliseconds just to transfer the connection state with no (empty) socket buffers. The size of such an idle connection state is about 500 bytes which includes the socket data structure and the TCP control block for the two (MH side and FH side) sockets plus some control information. In all the other cases, the handoff times were measured with the FH pumping data as fast as possible, given that the maximum window size for both parts of the I-TCP connection was equal to the socket buffer size. The handoff time increases with the size of socket buffers. This is because of the large difference in the bandwidths of the wireless and the wired media (2 Mbps WaveLAN versus 10 Mbps Ethernet) which causes the socket buffers at the MSR *to be full most of the time.* Thus when the MH switches cells, I-TCP handoff involves transferring a full send buffer for the MH side socket and a full receive buffer for the FH side socket from one MSR to another. The I-TCP handoff in the case with 32 Kbyte socket buffers can add up to 64 Kbytes (and possibly some more due to any pending user level buffers) to the basic connection state of about 500 bytes.

We repeated the above experiments with two other cell configurations i.e. - *i)* overlapped cells and *ii)* non-overlapped cells which are not adjacent. The second configuration was simulated by the MH staying disconnected for 2 seconds before it switched to a new cell. We did not find any significant difference in the handoff time in both these configurations compared to our test configuration of adjacent non-overlapped cells mentioned above.

We further analyzed the I-TCP handoffs to determine how much time is spent in each step. The results of our analysis are summarized in figure 7 for two representative cases - *a)* idle connection and *b)* active connection with 32 Kbyte socket buffers. All timing data shown is with reference to the instant when a greeting is received by the MSR from an MH entering its cell. In the case with idle connection, the new MSR took about 60 msecs to establish the socket skeletons from the information contained in the MH greeting message after which it sends a forward pointer (MICP_FWDPTR) to the old MSR. This included the time for two context switches — from the msrmicp process which receives the MH greeting to I-TCP daemon which creates the skeleton sockets and back to msrmicp which then sends the forwarding pointer. It took additional 150 msecs before the handoff request arrives at the new MSR. The actual state transfer took only 55 msecs which included two control messages and the corresponding ACKs (one to identify the MH, and another to identify the connection) in addition to two SIOCSETSTATE calls by the new MSR.

In the second case, with 32 Kbyte socket buffers, we see that after the handoff request is received by the new MSR, another 1230 msecs elapsed before the handoff was completed. Out of these, the state of MH

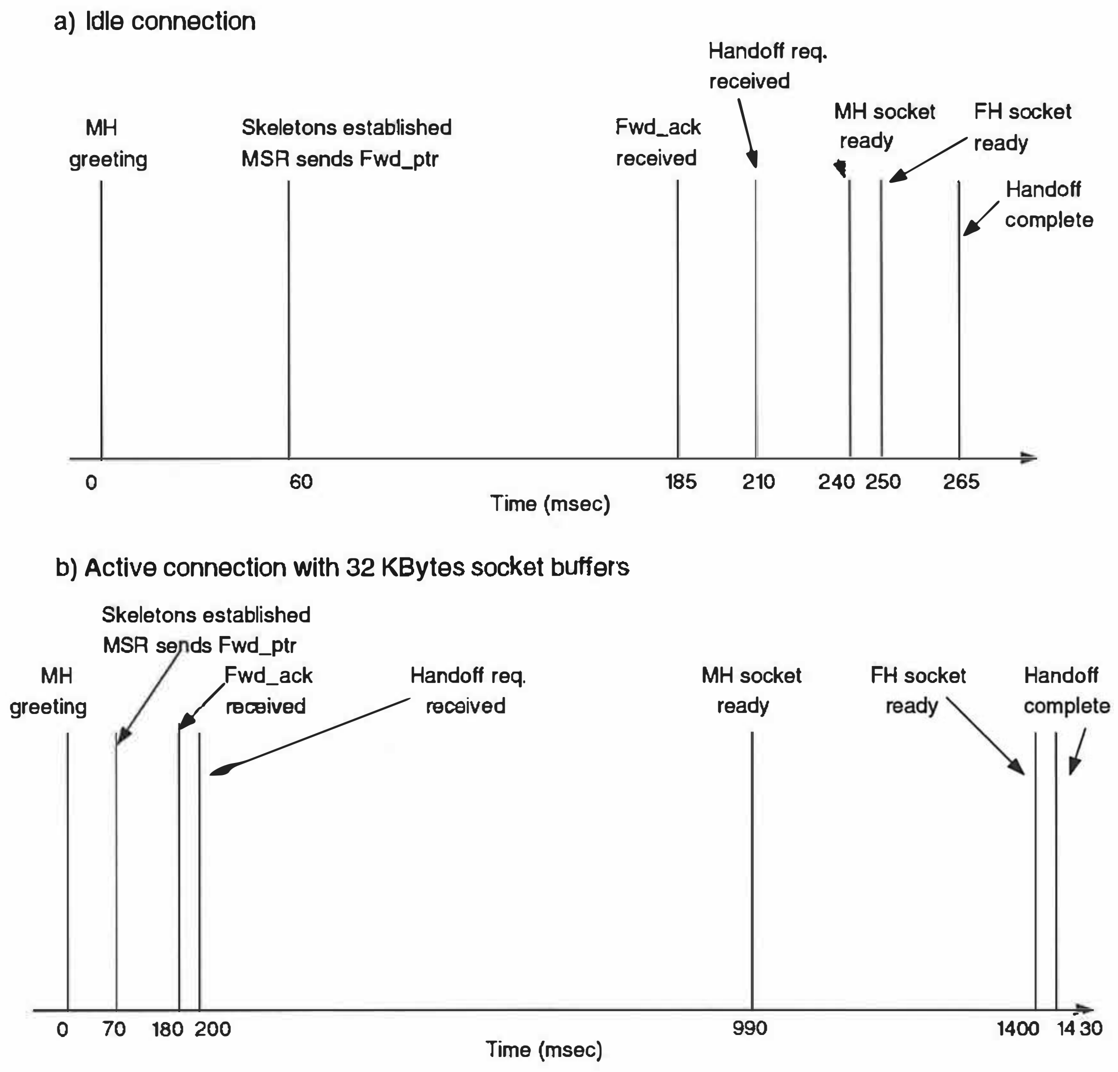


Figure 7. Analysis of I-TCP handoff events

side socket was transferred in 790 msecs whereas the state of FH side socket took 410 msecs. Thus with large socket buffer sizes, a major part of handoff time is spent in transferring the buffered data. The time to transfer MH side socket is larger than the corresponding time for the FH side socket mainly because the TCP handoff connection between the two MSRs is still in a *slow start* phase while transferring the MH side socket state.

From the above figure, state transfer of each idle I-TCP connection needs approximately 50 msecs, after a handoff connection is established between the MSRs; whereas transferring the state of a connection that was active takes 1 to 1.5 seconds for large socket buffer sizes. It should be possible to reduce the handoff time with pre-established handoff connections between MSRs and faster context switches between the MSR daemon processes.

6 Example Applications

We have used I-TCP to improve the performance of TCP based applications running on mobile wireless machines in our indoor wireless LAN environment. Some of these applications are listed below along with a description of the changes needed for I-TCP:

1. **ttcp benchmark** — We used the **ttcp** benchmark to perform the handoff experiments reported in the previous section. The regular socket calls for *connect, listen* and *accept* were replaced by their I-TCP equivalents. Throughput experiments conducted using the **ttcp** benchmark comparing the performance of I-TCP with that of regular TCP have been reported in [5].
2. **ftp** — **ftp** uses two kinds of TCP connections - one for control and another for data. The control

connection is used to exchange application layer control messages between the ftp client and server while a separate data connection is opened for every file transfer. We developed a *hybrid* ftp which we call i-ftp to use I-TCP for data connections but regular TCP for control connections. Using I-TCP for the data connections affords us with throughput advantages over regular TCP while the end-to-end nature of the control connection which uses regular TCP gives us adequate error detection for MSR failures which may cause loss of I-TCP (data) connections.

3. **Chimera WWW browser** — Chimera[5] uses HTTP to access WWW services over the Internet which is based on TCP. We modified the Chimera sources to use I-TCP instead of TCP which considerably improved the time needed to download large files at a mobile host.

7 Related Work

Performance improvements in the end-to-end transport layer throughput with the indirect model have also been reported elsewhere using a separate transport protocol for the wireless link [29]. Termination of the wireless protocol stack at an intermediate base station has also been explored in [15] for a digital cellular telephone network. A scheme for installing filters at the MSRs under MH control is described in [30]. In addition, the indirect approach where the intermediary (or a set of intermediaries) exists at higher protocol layers has been investigated by [2, 3].

Link layer retransmissions can be used on error-prone wireless links to bring their error rate on par with that on the wired networks but such an approach interferes with the end-to-end retransmissions of TCP and thus may not result in improved performance [8]. *Fast retransmissions* coupled with modifications to the TCP software on the mobile hosts [6] solve only part of the problem since most TCP implementations on the fixed network do not support fast retransmissions. I-TCP is based on an *indirect protocol model* proposed in [4] which takes a more general approach towards host mobility and wireless links.

8 Conclusion

In this paper we have described the implementation of system support for handing off active transport layer connection state in mobile environments. Such support is essential for indirect protocols such as I-TCP which show improved performance over conventional end-to-end protocols for wireless mobile hosts. The system support consists of daemon processes at the MSRs managing indirect connections and handoffs and also the coordination with mobile IP handoff between the old and the new MSRs when a mobile host switches cells.

Our measurements with I-TCP handoffs show that operating system support in the form of suitable handoff mechanisms can help in achieving efficient transport layer handoffs. We also analyzed the handoff data to determine the time consuming activities in I-TCP handoffs. A kernel resident implementation of I-TCP, though less flexible than a user level implementation such as ours, should further cut down on the copying overhead incurred by I-TCP connections.

We also described TCP based applications that we ported to I-TCP resulting in improved performance. We are planning to build indirect presentation layer protocols over I-TCP. Partial support for building such protocols already exists in our I-TCP implementation in the form of remote *ioctl*. Using the remote *ioctl* on the indirect socket at the MSR, makes it possible for a mobile host to install a filter in either direction of the fully duplex stream of the I-TCP connection. Such a filter can also be used to send the data received from a fixed host to a higher layer at the MSR for further processing before it is sent to the mobile host over the wireless link.

Acknowledgments

We thank John Ioannidis for providing us with the source code of Columbia's Mobile-IP implementation. The WaveLAN driver for Mach used in our experiments was originally written by Anders Klemets and adapted for our network configuration by Girish Welling. We also thank the anonymous reviewers of the extended abstract for their valuable comments.

Availability

We plan to make the sources of our I-TCP implementation publicly available sometime during summer 1995.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: a new kernel foundation for UNIX development. In *Proc. of the USENIX 1986 Summer Conference*, July 1986.

[2] A. Athan and D. Duchamp. Agent-mediated message passing for constrained environments. In

[5] Chimera was developed by John Kilburg.

USENIX Symposium on Mobile and Location-Independent Computing, August 1993.

[3] B.R. Badrinath, A. Acharya, and T. Imielinski. Structuring distributed algorithms for mobile hosts. In *Proc. of the 14th Intl. Conf. on Distributed Computing Systems*, pages 21–28, June 1994.

[4] B.R. Badrinath, A. Bakre, T. Imielinski, and R. Marantz. Handling mobile clients: A case for indirect interaction. In *4th Workshop on Workstation Operating Systems*, October 1993.

[5] A. Bakre and B.R. Badrinath. I-TCP: Indirect TCP for mobile hosts. To appear in the *Proc. of the 15th Intl. Conf. on Distributed Computing Systems*, May 1995.

[6] R. Caceres and L. Iftode. The effects of mobility on reliable transport protocols. In *Proc. of the 14th Intl. Conf. on Distributed Computing Systems*, pages 12–20, June 1994.

[7] E.C. Cooper and R.P. Draves. C Threads. Technical Report CMU-CS-88–154, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, June 1988.

[8] A. DeSimone, M.C. Chuah, and O.C. Yue. Throughput performance of transport-layer protocols over wireless LANs. In *Proc. of Globecom '93*, December 1993.

[9] D.J. Farber, G.S. Delp, and T.M. Conte. A thinwire protocol for connecting personal computers to the Internet. *RFC 914*, September 1984.

[10] J. Ioannidis, D. Duchamp, and G.Q. Maguire. IP-based protocols for mobile internetworking. In *Proc. of ACM SIGCOMM*, pages 235–245, September 1991.

[11] J. Ioannidis and G.Q. Maguire. The design and implementation of a mobile internetworking architecture. In *Proc. of USENIX Winter Technical Conference*, January 1993.

[12] V. Jacobson. Congestion avoidance and control. In *Proc. of ACM SIGCOMM*, pages 314–329, August 1988.

[13] V. Jacobson. Compressing TCP/IP headers for low-speed serial links. *RFC 1144*, February 1990.

[14] V. Jacobson, R. Braden, and D. Borman. TCP extensions for high performance. *RFC 1323*, May 1992.

[15] M. Kojo, K. Raatikainen, and T. Alanko. Connecting mobile workstations to the Internet over a digital cellular telephone network. In *Mobidata Workshop on Mobile and Wireless Information Systems*, November 1994.

[16] S.J. Leffler, M.K. McKusick, M.J. Karels, and J.S. Quarterman. *The design and implementation of the 4.3BSD UNIX Operating System*. Addison Wesley, 1989.

[17] P. Manzoni, D. Ghosal, and G. Serazzi. Impact of mobility on TCP/IP: an integrated performance study. To appear in *the IEEE Journal on Selected Areas in Communications*, 1995.

[18] A. Myles and D. Skellern. Comparison of mobile host protocols for IP. *Journal of Internetworking Research and Experience*, 4(4), December 1993.

[19] J. Nagle. Congestion control in IP/TCP Internetworks. *RFC 896*, January 1984.

[20] J. Postel. User datagram protocol. *RFC 768*, August 1980.

[21] J. Postel. Transmission control protocol. *RFC 793*, September 1981.

[22] Y. Rekhter and C. Perkins. Optimal routing for mobile hosts using IP's loose source route option. *Internet Draft*, October 1992.

[23] J.H. Saltzer, D.P. Reed, and D.D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4), November 1984.

[24] W.T. Strayer, M.J. Lewis, and R.E. Cline Jr. XTP as a transport protocol for distributed parallel processing. In *Proc. of the USENIX Symposium on High-Speed Networking, Oakland, CA*, August 1994.

[25] W.T. Strayer, M.J. Lewis, and Jr. R.E. Cline. An object-oriented implementation of the Xpress Transfer Protocol. In *Proc. of the Second Intl. Workshop on Advanced Communications and Applications for High-Speed Networks (IWACA), Heidelberg, Germany*, September 1994.

[26] C. A. Thekkath, T.D. Nguyen, E. Moy, and E.D. Lazowska. Implementing network protocols at user level. In *Proc. of ACM SIGCOMM*, pages 64–73, September 1993.

[27] D. Velten, R. Hinden, and J. Sax. Reliable data protocol. *RFC 908*, July 1984.

[28] H. Wada, T. Yozawa, T. Ohnishi, and Y. Tanaka. Mobile computing environment based on internet packet forwarding. In *Proc. of the USENIX Winter Technical Conference*, January 1993.

[29] R. Yavatkar and N. Bhagwat. Improving end-to-end performance of TCP over mobile internetworks. In *IEEE Workshop on Mobile Computing*, December 1994.

[30] B. Zenel and D. Duchamp. Intelligent communication filtering for limited bandwidth environments. *To appear in the Fifth Workshop on Hot Topics in Operating Systems (HoTOS-V)*, May 1995.

A Wireless Adapter Architecture for Mobile Computing

John Trotter and Mark Cravatts
*AT&T Bell Laboratories,
600 Mountain Avenue,
Murray Hill,
NJ 07974*

Abstract

We present the architecture of an indoor wireless adapter card for use in nanocellular wireless networks. The card, called FAWN (Flexible Adapter for Wireless Networking), includes an RF modem, a CPU and an interface to a host computer. The card supports mobility, allowing portable machines to handoff between base-stations while maintaining a high speed, low latency wireless connection. By executing the MAC algorithms on the FAWN card the host computer is able to run its applications at full speed. In addition, this approach enhances the card's applicability for use in a wide variety of machines as it presents a simple and universal interface. The on-board CPU can also be used as the controller, allowing the card to be used as the central component in embedded wireless designs. The card has been constructed and is being used as part of a larger wireless project in AT&T.

Introduction

Advances in technology have enabled high speed wireless communication and this has spurred the growth of wireless, and now mobile computing systems. This technology has enabled many new wireless services and systems including a proliferation of wireless LANs which make the first step toward a mobile computing environment. We are constructing a wireless computing testbed to investigate how a mobile computing system will be architected, integrated into existing and future infrastructure and how applications that run on the system will interact with the machine and other applications.

Several models for mobile and wireless computing have been proposed. Xerox's tab [8] uses a small hand-held device, called a tab, as a terminal which then interacts with the surrounding network of machines which contain the intelligence. This allows users to pick up and use an arbitrary tab and immediately have access to their environment. The tab system uses an infra-red link for data receive and transmit, and allows communication in office size cells which are connected to the installed backbone network. The Infopad system [5,9] uses a similar approach of a terminal device, this time connected using a higher speed RF modem to access interactive data. The Infopad also relies on the network to provide the resources, acting as a terminal for the data.

Intermediate approaches split the intelligence between the device and the network. Some of the computation can be carried out using powerful processing resources in the backbone system instead of on the mobile which has limited processing resources as well as a limited power budget. Applications take advantage of as much communications bandwidth as is available by adaptively altering the amount of processing on the backbone network verses the portable machine.

A third approach has the intelligence in the mobile device and uses the network to access other devices on a peer to peer basis. This approach is similar to the current model of networked computing, and this model is supported by wireless LAN systems running Mobile IP [6]. Mobile IP allows the definition of a "mobile subnet" which has many mobiles associated with it. When communicating with a mobile machine, data is sent to any one of several fixed hosts associated with that subnet. These hosts either know where the mobile is (i.e. which base station the mobile is communicating with), or can find out by asking a set of other base stations. The WaveLAN system [10] is a wireless LAN system that allows wireless extension of existing Ethernet networks, and has been used as the physical layer for several mobile computing projects.

We envisage a model of wireless untethered computing which allows continual connection of mobile machines to the network as users move around their office, corridors and conference rooms. The system will support several models of access, from terminals to intelligent mobile hosts. In order to support this model, portable machines must be equipped with suitable wireless interfaces, wireless base stations must be installed and the backbone network must be enhanced to support mobile users [1,5,7,11]. Our objective is to develop an indoor wireless computing environment that supports continual access by heterogeneous end systems in the presence of mobility, supports location aware applications and supports end-to-end performance requirements over the network. In order to be able to investigate these issues we needed to have complete control over the software and hardware of the wireless interface. Existing wireless designs were unsuitable for our experiments so we built our own hardware which allowed us to experiment with innovative handoff and MAC schemes. We are also able to use this approach to experiment with applications that are split between the mobile and the backbone network.

The System

A high level view of our system is shown in Figure 1. The model uses a wide area ATM backbone network, connected to local area communication fabrics which, in turn, connect to endpoints. The endpoint machines can be base-stations which provide the wireless hardware necessary to support a final hop to a mobile, a service provider such as a computation server or database server or a user's workstation. The complete system is called SWAN (Seamless Wireless ATM Network) [1]. A mobile computer can connect to the base-station using a wireless adapter, as can "personal terminals" which are devices that act as terminals onto the network. Each base-station normally provides an access point for a few offices, although cells may be smaller or larger, depending on the demand in that area. For example, a conference room may have two or more cells to accommodate many users in a small area, or the cells may be far larger in areas of low use such as corridors or outdoor walkways. A mobile host is able to roam around while maintaining connectivity, handing off between base-stations on the network even if the base-stations are on different local fabrics. Thus we support ubiquitous mobile computing anywhere there is a base station. If a mobile becomes disconnected for any reason it can reconnect once it moves within range of a base station. In addition to communicating using wired base stations, a group of users can set up a temporary network so they can exchange files or use a service provided by one of the hosts.

One of our main objectives is to support multimedia access over the network for the mobile systems. An important part of being able to provision multimedia services is being able to provide some form of Quality of Service (QoS) guarantee for applications that use the network. Asynchronous Transfer Modem (ATM) is expected to be able to satisfy these requirements by providing per Virtual Channel (VC) performance guarantees. Research investigating per VC QoS can be extended to the wireless domain if ATM is used from end-to-end. In addition, an ATM interface allows a mobile host to make informed packet routing decisions on to the wireless hop, whose available bandwidth will probably be lower that the wired backbone and may suddenly change. The QoS model also allows renegotiation of the bitrate of each VC which supports the changing bitrate which may well occur in a wireless system. By directly routing ATM packets through the base station we avoid any protocol conversion or data repackaging which may of been necessary if a proxy at the base-station intercepted the data. Therefore, we chose to use ATM as our backbone network as well as for the wireless hop.

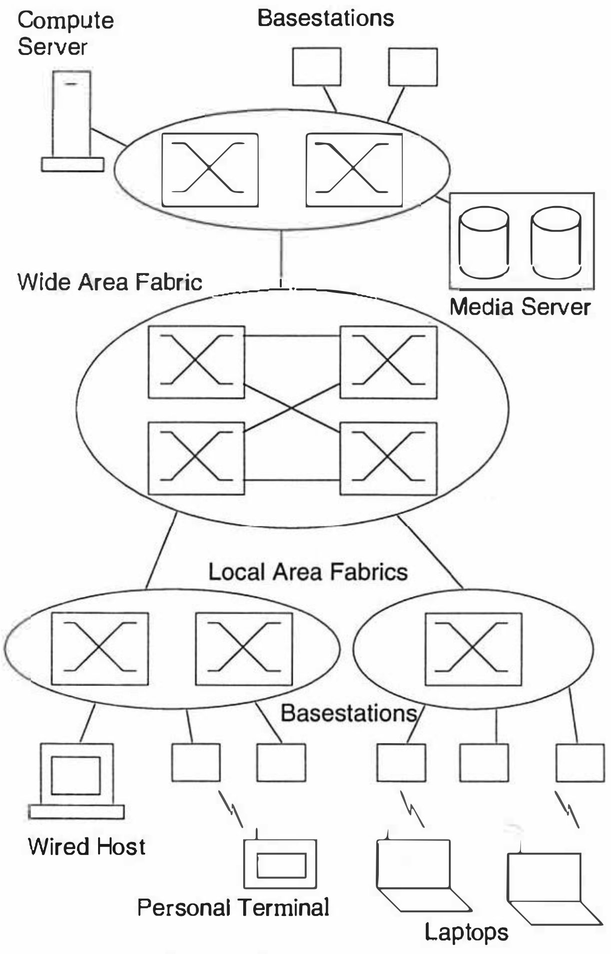


Figure 1. System overview.

To fully support multimedia traffic the network has to appear as seamless as possible. During handoff the data stream has to be rerouted from the old base-station to the new one with the minimum of loss so that the network looks continuous to machines using it. An important feature in our mobile system is the ability to handoff at high speed. This minimizes cell loss and allows for a high bitrate to be guaranteed and maintained when a VC is established. We use mobile directed handoff which allows the mobile to control the handoff. When the received power is too low the mobile begins searching for a better base-station, and if it finds one, it initiates a handoff. Because both the base-stations and the mobile are all informed during the move, the handoff processes can be carefully controlled to ensure that no data is lost in the transition.

Several other approaches to mobile computing use a direct sequence spread spectrum modems such as WaveLAN. While newer versions of the WaveLAN system offer up to 20Mbits/sec, this bandwidth has to be shared among many users. The problem of providing a guaranteed bitrate is complicated because the protocol does not allow guaranteed access to the channel. Other spreading codes can be used to increase the number of channels, but the aggregate bandwidth available using this technique is less than using a Frequency Hoping Spread Spectrum (FHSS) modem [4]. We chose to couple a MAC protocol (based on token passing) which guarantees channel access with a FHSS modem in order to support a high aggregate bandwidth. The modem we use gives us up to 1M bits/sec over the wireless link in the 2.4Ghz ISM band, allowing each channel a guaranteed bitrate. Each mobile is assigned one virtual channel, and several modems allow many channels to be used simultaneously up to a limit imposed by the RF bandwidth. If even more channels are needed they can be time division multiplexed onto a single modem's channel.

In our system we consider a nanocell indoor wireless environment with fixed base stations that are interconnected with an ATM network. ATM packets are used end-to-end in the network, and the packets are encapsulated for the final hop from the base station to the mobile unit. The system supports mobile directed handoff from one base station to another, which is done at very high speed and with low latency as part of the MAC layer.

The Architecture

In order to support the above requirements for multimedia the adapter card had to be able to handle ATM data, support fast handoff and provide the wireless bandwidth needed. The adapter card, called FAWN (Flexible Adapter for Wireless Networking) provides the necessary computing power to run the MAC and handoff software and provides an interface between the host computer and the RF modem. To enhance reuse, the FAWN adapter was designed so that it could be used as the wireless adapter card in either a base-station, a mobile or an embedded application such as a personal terminal. In order to ensure the widest compatibility with desktop and portable computers we chose to use a PCMCIA interface for the FAWN adapter.

The FAWN adapter set consists of two cards, a modem card and a CPU card, and several interfaces. The CPU card provides the CPU, data and program memory as well a buffer for data communication between the CPU and the PCMCIA interface, or between other processes running on the system. It also provides an interface for the modem card, a PCMCIA host and a peripheral card which can be used by other systems such as a personal terminal [3] or embedded base station. The CPU card can accept one or more modem cards which provide hardware for packet buffering, clock recovery and the interface to an RF modem

The FAWN adapter set can be used in one of several configurations. The base station configuration is shown in Figure 2. The base station controller is a computer which has an interface to the backbone ATM network as well as an expansion bus which is used to plug in several PCMCIA adapters. Each PCMCIA adapter accepts a FAWN CPU card which in turn interfaces with several modem cards. Because we use a FHSS modem, one modem card is required per channel, therefore so that the base station can provide enough channels for high demand areas several modem cards must be supported.

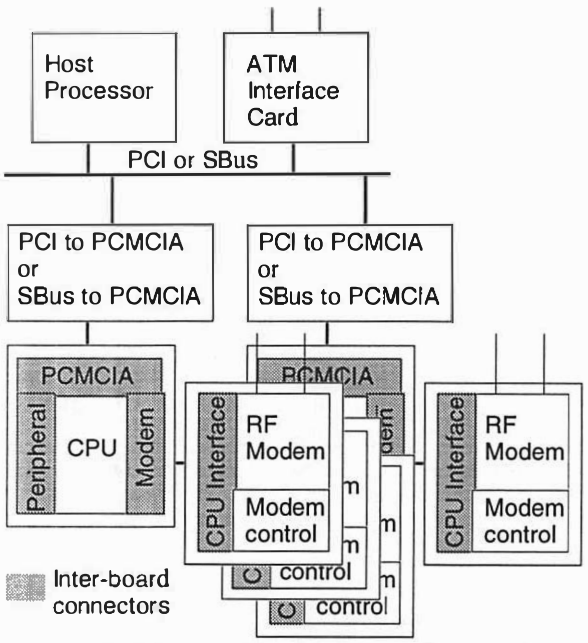


Figure 2. The FAWN card used in a base station

When used as an adapter for a mobile system, the FAWN card interfaces to one modem card and the PCMCIA connector in the mobile host. This configuration is shown in Figure 3.

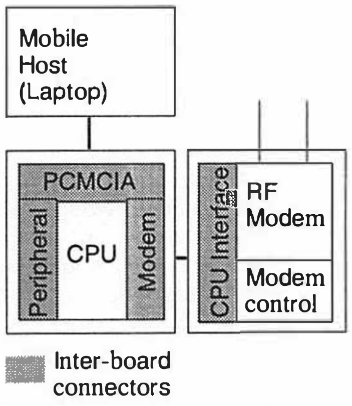


Figure 3. The FAWN card used in a mobile host

When the FAWN cards are used as an embedded base station the peripheral interface is used to support a network interface, such as ATM. This configuration is shown in Figure 4. One or more modem cards can be plugged into the CPU card, supporting fewer distinct channels than the base station. This version of the card can be used in areas of medium demand, like in an office or corridor

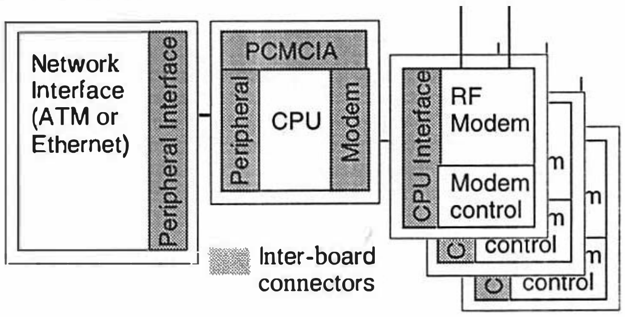


Figure 4. The FAWN card used as an embedded base station

Another incarnation is a personal terminal [3]. Here a peripheral with video and audio I/O capability is plugged into the peripheral connector allowing the FAWN card to provide the wireless link. In addition the FAWN card provides the necessary CPU to run the terminal's application. Here we take advantage of the card's ability to multitask, so that the MAC layer can run on the same CPU as the PDA application.

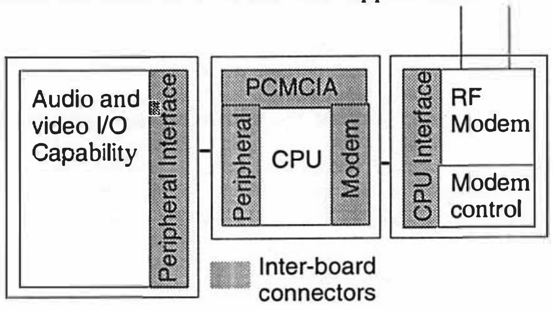


Figure 5. The FAWN board used in a personal terminal

Detail of the FAWN card is shown in Figure 6. The main parts are the PCMCIA interface that provides connectivity to the host machine, the CPU and RAM subsystem which provides the raw processing power, the dual port RAM system which provides a fast way of transferring information from the host CPU to the adapter card, and the modem interface which provides packet buffering and provides the logic to interface with the RF modem.

The following sections describe each part of the architecture in detail:

- PCMCIA Interface. The PCMCIA interface is implemented in an FPGA and provides an interface between the host computer and the adapter card. In normal operation, the interface allows the host machine to access control registers that configure the interrupts for the CPU on the card and control the reset, as well as providing access to the Dual Port RAM (DPR). The PCMCIA interface also allows the host computer to access the peripherals that the FAWN card's CPU would normally access by stalling the CPU, these include the other side of the DPR as well as the memory, the RF modem interface and the real time clock.
- Dual Port RAM (DPR). The DPR provides a high speed interface between the host computer and the adapter card. The DPR is used to implement the queues necessary for communication between the host and FAWN. By using a RAM structure (as opposed to a FIFO) the implementation of arbitrary queue structures with differing sizes and priorities is easy. When FAWN is used as an embedded controller, communication between the MAC process and the higher level processes still continues using the DPR, allowing a standard interface to be presented to all applications, wherever they run. The DPR provides support for semaphores to ensure that two identical locations are never accessed at the same time by the host and FAWN. 32K bytes of DPR are used in the prototype. The DPR also facilitates the conversion of the 32 bit word used by FAWN's CPU to the 16 bit word needed by the PCMCIA interface.
- The adapter card contains a 32 bit RISC CPU, the ARM 610 [2] running at 20MHz. This microprocessor was chosen because of its low power as well as it being a powerful 32 bit CPU. The prototype systems includes 4M bytes of memory for program and data and interfaces to another FPGA which controls the modem and provides buffering for the data packets. The ARM610 is powerful enough so that a single CPU can control many modem boards as well as provide the CPU to control a peripheral board which

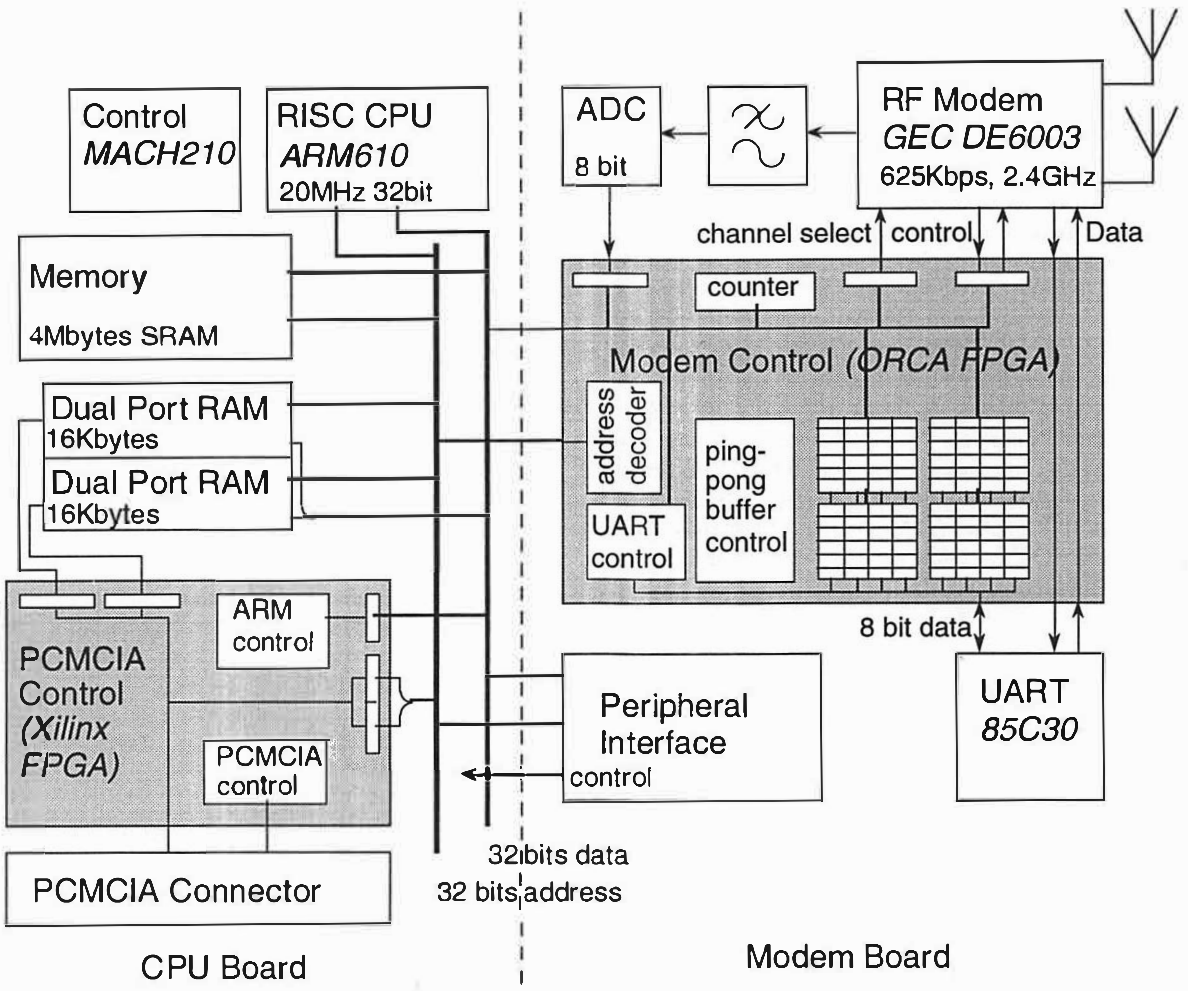


Figure 6. Detailed architecture of the FAWN board

contains an external network interface or other circuits. The CPU is also used to execute Forward Error Correction (FEC) code which ensures that erroneous cells are at least detected if they cannot be corrected.

- RF Modem Interface. The modem interface is implemented in an FPGA and includes a packet buffer and a hop controller. The packet buffer is 64bytes long and allows the buffering of a complete ATM packet as well as extra space for encapsulation and error control bits. The FPGA implements four such buffers, two for transmit and two for receive. During a receive operation one of the two receive buffers slowly fills up (at the data rate). When the buffer is full an interrupt is generated so that the CPU can empty the buffer all at once, meanwhile the other buffer begins to fill from the data stream. During transmit operation the CPU fills a transmit buffer then sets a bit to tell the FPGA that the buffer is ready to be sent. The buffer is then made available to the UART at the data rate. Once the buffer has been sent the FPGA generates an interrupt so that the CPU knows that the buffer is now available to be filled. Meanwhile the second transmit buffer can be sent. The provision of these buffers allows the CPU to be decoupled from the low level byte based transceive operations. The FPGA also provides the necessary control logic for all the modem control including hop control (selecting the channel for the modem). Because we use an FPGA we are able to interface with other faster modems as they become available.
- The modem FPGA also provides a real time clock which can be read and reset by the CPU as well as being able to interrupt the CPU when a programmed time has been reached. In addition, there is an interface to an Analog to Digital Converter (ADC) which converts the received signal strength to a digital value and makes this available to the CPU. The ADC can also interrupt the CPU once a threshold signal level has been crossed, this facilitates interrupt driven handoff. The FPGA also provides an interface to a UART which takes the raw bitstream from the modem and converts it to 8bit data bytes. We use SDLC encoding, which is a synchronous protocol that

guarantees that there are enough transitions to ensure that the UART remains synchronized.

- RF Modem. We use a GEC Plessey frequency hopping spread spectrum modem (DE6003) which operates in the 2.4Ghz ISM (Industrial, Scientific and Medical) band. The modem uses gaussian minimum shift keying as the modulation technique and the parts we have operate at a bitrate of 625K bits/sec (future pin-replaceable modems will operate at 1M bits/sec). The modem can operate at one of 83 channels, assigned for use on this channel by the FCC. The RF modem has two antenna ports for antenna diversity, so that the antenna with the best signal strength can be chosen during receive.
- Peripheral Interface. The design includes a peripheral interface that allows additional peripherals to be plugged in. Examples include an external Ethernet or ATM adapter or a display with an audio I/O device allowing the card to be used in a personal terminal.

Software

The software is split between the FAWN adapter and the host CPU, a representation is shown in Figure 7. When FAWN is used in an embedded system all the software runs on the CPU as different processes. To enable this multiprocess software structure, the FAWN system runs a basic kernel which provides support for threads and schedules the MAC tasks as well as the queuing scheme for the ATM packets. The CPU also executes forward error correction, error detecting and correcting packets of data. Because the card runs a kernel, it is simple to implement another thread which might be responsible for running ATM adaptation layer code. By using the FAWN board to execute the bulk of the code we are able to present a standard interface and therefore improve compatibility over the many different platforms that support PCMCIA. Running the time-critical code on the adapter ensures that the data is handled in a timely manner.

The MAC software runs as a high priority interrupt driven process on the FAWN CPU. On receiving an interrupt the MAC retrieves cells from the packet buffer, error checks and corrects the cells, then places the cell in the DPR interface. The queue manager operates using interrupts from the host and organizes packets in the DPR so they can be read by the host ATM interface. When FAWN is used with a host CPU, the host CPU reads the packets from the PCMCIA interface and executes the adaptation layer which makes the cells available to the application (in the case of the mobile host) or the ATM interface card. If the process that uses the cells runs on FAWN then it runs as another process thread and communicates with the queue management software using software interrupts.

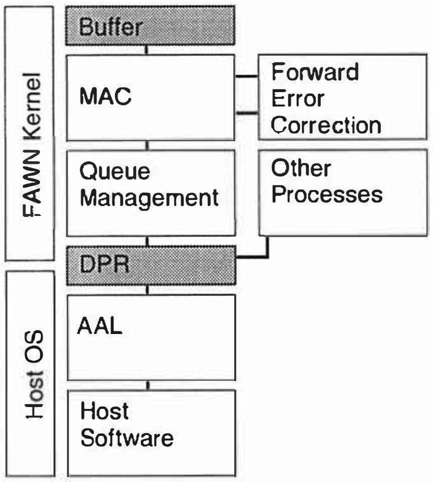


Figure 7. The software architecture

The system that uses the FAWN card, SWAN, also implements mobile directed handoff. Therefore the MAC interface is able to signal the base station when a move is about to occur, the old base station can then reroute the data stream to the new base station.

Implementation

The FAWN card is implemented as two PC boards, the modem board and the CPU board that plug together, these are shown in Figure 8. The two boards are designed to fit in the floppy drive bay of an AT&T GIS 3181 laptop computer, which is the machine we are using as the mobiles in our prototype. Eventually we plan to implement the design in a type III PCMCIA card that can be used in a wider variety of portable machines. Several modem cards can be plugged into the CPU card for use in a base station, and several CPU cards can be plugged into PCMCIA slots in the host computer. We are currently using Pentium based PCs as the base stations, and these interface with the ATM interface card, currently from FORE. The prototype is installed in our office environment and allows users of mobile systems to remain connected to the backbone network as they move.

Conclusion

In this paper we have described the architecture of an adapter card designed to support wireless mobile computing. We architected and built the card which is now in use in a prototype indoor wireless network. The card was architected and designed in a very flexible way, allowing the same card to be reused in base stations and mobiles. We are able to support high speed interfaces from the host machine to the adapter card using the DPR. Because the FPGA handles the byte level communication operations and only presents the CPU with complete packets the CPU can be left to run MAC level code and still have enough capacity to execute embedded application programs. The MAC and handoff design is aided by the interrupt driven ADC and the real time clock which are available on the card. The processor on the card is responsible for all the low level operations, simplifying the interface presented over the PCMCIA interface which in turn simplifies the software that has to be implemented on the host. This increases the ease with which the system can be integrated with new hosts.

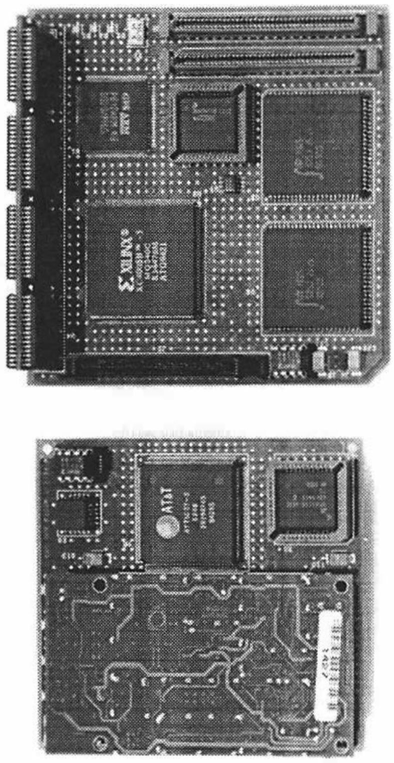


Figure 8. The CPU board and Modem board that make up the FAWN set.

References

[1] P. Agrawal, A. Asthana, M. Cravatts, E. Hyden, P. Krzyzanowski, P. Mishra, B. Narendran, P. Mishra and J. Trotter. "A Testbed for Mobile Networked Computing," Proceedings IEEE Communication Conference, Seattle, WA, June 1995

[2] ARM610 Data Sheet, Publication No DS3554, Issue No 2, March 1993 GEC Plessey Semiconductors, Sequoia Research Park, 1500 Green Hills Road, Scotts Valley, CA 95066

[3] A. Asthana, M. Cravatts and P. Krzyzanowski, "An Indoor Wireless System for Personalized Shopping Assistance," Proc. IEEE Workshop on Mobile Computing Systems and Applications, Santa Cruz, CA. December 1994.

[4] D. Bantz and F. Bauchot. "Wireless LAN Design Alternatives" IEEE Network March/April 1994 Vol. 8 No. 2, pp.43 - 53

[5] B. Barringer, T. Burd, et. al. "Infopad: A System Design for Portable Multimedia Access." In Wireless 1994, Calgary, Canada, July 1994.

[6] J. Ioannidis, D. Duchamp and G.Q. Maguire. "IP-based Protocols for Mobile Internetworking" Proc SIGCOMM 91, 1991 pp. 234-245.

[7] K. Keeton, B. Mah, S. Seshan, R. Katz, and D. Ferrari. "Providing Connection- Oriented Services to Mobile Hosts." In Proceedings of the USENIX Symposium on Mobile and Location-Independent Computing, pages 83-102, Cambridge, Massachusetts, August 1993.

[8] B. Schilit, N. Adams, R. Gold, M. Tso and R. Want, "The PARCTAB Mobile Computing System," Proceedings of the Fourth Workshop on Workstation Operating Systems. Napa, CA, October 1993, pp. 34-39

[9] S. Sheng, A. Chandrakasan and R.W. Brodersen, "A Portable Multimedia Terminal," IEEE Communications Magazine, December 1992, pp. 64-67

[10] B. Tuch, "Development of WaveLAN(tm), an ISM Band Wireless LAN," AT&T Technical Journal, July/August 1993, pp 27-37

[11] M. Weiser. "Some Computer Science Issues in Ubiquitous Computing." Communications of the ACM, 36(7):209-220, July 1993.

MCE: An Integrated Mobile Computing Environment and Simulation Testbed[*]

Ramki Rajagopalan[†]
Bell Northern Research
2201 Lakeside Blvd
Richardson, TX 75082-4399
ramki@bnr.ca

Sridhar Alagar and S. Venkatesan
Computer Science Program, EC 31
The University of Texas at Dallas
Richardson, TX 75083-0688
{sridhar, venky}@utdallas.edu

Abstract

Integrated Mobile Computing Environment and simulation testbed (MCE) is a software platform to design, develop, and test algorithms and applications for a network of mobile and static computers. MCE is designed with the idea of providing users with a transparent system so that an application may be tested on the simulation testbed and then used directly on the network without any source code changes. MCE provides default functionalities like routing, location tracking, and handoff along with service access points to add or change services resulting in a unique integrated environment for development and testing. MCE may also be used as a platform to evaluate and compare various mobile network algorithms.

1 Introduction

MCE is a software platform for a network of mobile and static computers on which various protocols, algorithms and applications can be built, tested, evaluated and supported. When mobile systems have a large number of users, it is not practical to test applications with real mobile systems. Conventional simulators like event driven simulators suffer from the problems that they are not close to reality and, when the number of mobile users is large, the time taken to test an application is enormous. Hence, we also provide facilities within MCE so that it can be used as a simulation testbed to test protocols developed for a mobile network.

MCE simulates a mobile host by a process, and mobility is accomplished by checkpointing and process migration. In this aspect, MCE differs from other platforms such as the one described in [5]. This approach also provides for a transparent platform where the application developer need not make any changes when moving from the simulation testbed to the actual mobile environment.

MCE provides features such as routing, location tracking and multicasting as default features. We have taken a modular design approach so that any communication protocol [3, 4, 6, 7] may be integrated into MCE and evaluated. Currently, we have our initial version of MCE simulation testbed running in our lab. The design and implementation details are described in Section 7. The current status report is given in Section 8. We are planning on setting up an *ftp* site for access to MCE information, and these details are given in Section 9.

2 Mobile Computing Systems

A mobile computing system consists of a network of *static hosts* and *mobile hosts*. A *static host* has a fixed geographical location in the network and a fixed network address. A *mobile host* can move and has no fixed location. However, every *mobile host* has a unique network address. All the *static hosts* are part of a wired network that forms the backbone of the mobile computing network. Some of the *static hosts* may be designated as *Mobile Support Stations* (MSS). An MSS has the necessary infrastructures to support *mobile hosts* and related functionalities. Each MSS supports *mobile hosts* within a certain geographical area called the *cell*. A *mobile host* may belong to at most one cell at any time.

Communication among MSSs is supported by using the backbone network. We assume that a *logical* communication channel exists between every pair of MSSs. Communications between an MSS and *mo-*

[*]This research was supported in part by NSF under Grant No. CCR-9110177 and by the Texas Advanced Technology Program under Grant No. 9741-036.

[†]This work is being done as part of author's graduate work at The University of Texas at Dallas

bile hosts in its *cell* are over a wireless medium. All communications among the *mobile hosts* are through their MSSs. Each *mobile host* has an MSS designated as the *home MSS* where all relevant data for the *mobile host*, such as the current location, subscribed services etc., are maintained.

A *mobile host* may migrate from one cell to another at any time. In order to keep track of the number and identity of *mobile hosts* in its *cell*, an MSS periodically broadcasts an MSS specific *beacon* [6]. When a *mobile host* h receives the *beacon* from MSS s_{new}, h verifies the MSS address; if the MSS has changed from what h remembers, say s_{prev}, then h sends a response to s_{new} that includes the address of s_{prev}. When s_{new} receives this message, a *handoff* procedure is executed between s_{new} and s_{prev}.

3 MCE Architecture

MCE consists of several networked UNIX machines. We consider every *static* machine as a mobile support station and each MSS will have a server process, MSS Daemon (**mssd**), running in the background on the MSS. These daemon processes will control all communications that occur within the mobile network. We use *sockets* for establishing and controlling various communication interfaces between *static* hosts. In our integrated platform, we provide software to support real mobile hosts along with features to support mobile host simulation.

In case of simulation, we can develop and test applications for a mobile environment without the presence of actual mobile hosts and wireless interfaces. Each mobile host h in the cell of an MSS s_1 is simulated by a process p_h running on the MSS s_1. Migration of MH h from an MSS s_1 to another MSS s_2 is simulated by checkpointing and process migration of p_h from s_1 to s_2 followed by a handoff. The wireless link between an MSS and an MH is simulated using sockets. The messages sent along the sockets are delayed to match the bandwidth capabilities of wireless links. The mobility of MHs will be controlled by a mobility simulator (**msim**) process running on MSS machines. The correspondence between some of the entities in a mobile network and in the simulation is shown in Table 1. The **msims** communicate among themselves using sockets. The **msims** communicate with the MH processes (to checkpoint and migrate) using signals. A pictorial representation of the architecture is shown in Figure 1. Although, MCE is defined on the basis of networked UNIX machines, we have taken sufficient care to ensure that MCE could be ported easily to different hardware plaforms and operating systems. This has been done to make it easy to support MCE applications on different PDAs, notebooks etc., that become available in the market.

Mobile Network	Simulation
Mobile Hosts	Processes
Wireless Links	Sockets
Mobility	Process Migration

Table 1: Correspondence between Mobile Network and Simulation in MCE

In order to support actual mobile network application development, MCE provides two libraries with identical Application Programming Interfaces (**API**), namely, **MCE Simulation Library** and **MCE Mobile Environment Library**. Application developers may use the following steps to develop, test and deploy their mobile applications within MCE.

- Develop the application using the APIs provided by MCE.
- Link the application with **MCE Simulation Library** and get an executable for the simulation testbed.
- Test the application on the simulation testbed.
- Once the application has been debugged, create a new executable by linking the application to the **MCE Mobile Environment Library**.
- The application is now ready for use on a mobile host.

MCE application developers need not be aware of any OS/hardware dependencies within MCE as the MCE API resides on top of the operating system. Thus, MCE application developers need not be concerned with any machine and/or operating system dependencies within MCE. MCE does not require any modifications to the operating system. Moreover, MCE-specific processes such as **mssd** may be run as user level processes. This makes it easy to install MCE and develop and test MCE-based applications. This view of MCE is illustrated in Figure 2.

In the initial version of MCE, we have provided only the simulation and testbed features and hence **MCE Simulation Library** is the only library available at this time. We provide details about the simulation testbed environment in the following sections.

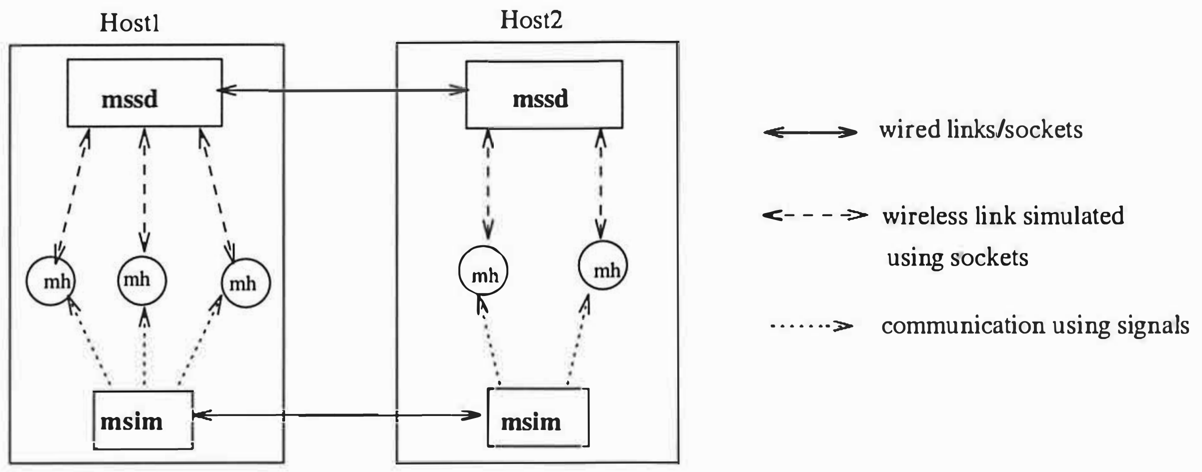

Host1
Host2
mssd
mssd
mh
mh
mh
mh
mh
msim
msim
wired links/sockets
wireless link simulated using sockets
communication using signals

Figure 1: The Simulation Testbed

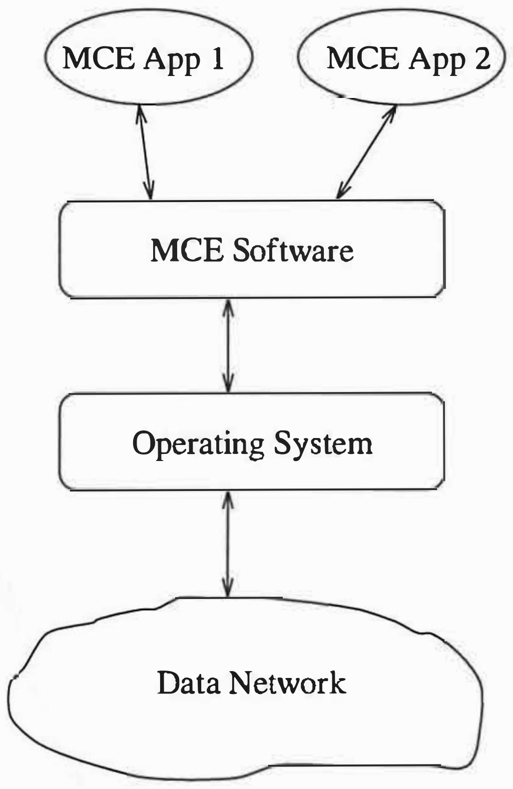


Figure 2: MCE Software on a System

4 Mobility Management

We use checkpointing to simulate mobile host migration from one MSS to another. Complexities introduced by checkpointing and process migration can be avoided by letting a single host simulate multiple MSSs. We chose not to take that approach for several reasons, some of which are:

- One of our design goals is to minimize development that is not common to both the simulation testbed and real mobile network within MCE.
- If mobile host processes were the only elements that were different from actual mobile computing network, then the performance of the mobile applications and algorithms can be measured effectively. The only difference we need to account for would be the mobile host processing and the wireless interface.
- By using the backbone network as part of the simulation, the static network part of MCE gets tested completely. Moreover, this lets us use an MSS as a simulation node as well as a real MSS supporting mobile hosts. This facility may help in testing and evaluating performance of either end involved in a protocol independently.

In the simulation testbed, when a mobile host enters an MSS, the corresponding **msim** process will decide when the mobile host will leave that MSS.

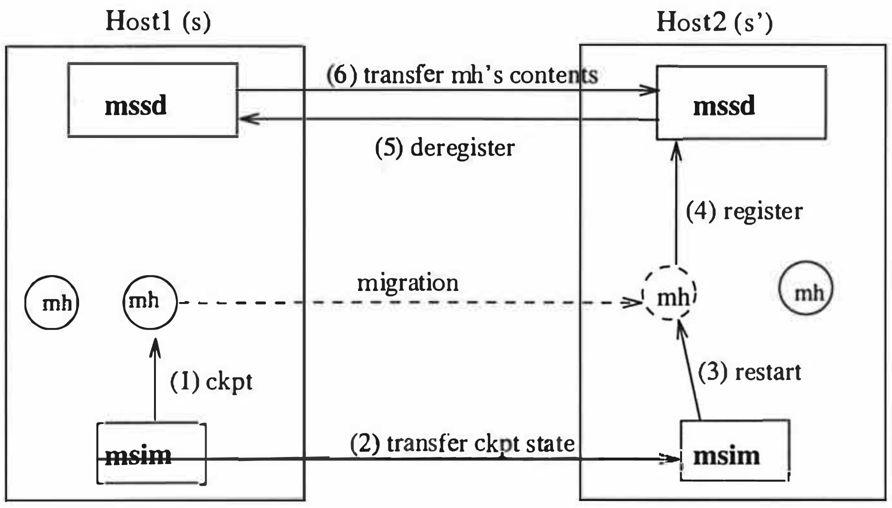


Figure 3: Mobility and Handoff

We use a simple scheme that assumes the time of stay in an MSS to be random (uniform distribution) and choose a destination. More sophisticated schemes may be designed based on the velocity of a mobile host and direction of its movement. For such schemes to work, **msim** should be aware of the topology and dimensions of the cells. We also provide an option to let users specify mobility patterns.

Let MH h move from the cell of MSS s to the cell of MSS s'. Let m and m', respectively, be the **msim** processes in the machines s and s'. The steps involved in the migration of MH h from s to s' are as follows.

- **msim** m sends a signal to mobile host h to checkpoint h's state ((1) in Figure 3).
- **msim** m then transfers the checkpointed state of mobile host h to **msim** m' in the MSS s' ((2) in Figure 3).
- **msim** m' then restarts the MH h from its checkpointed state ((3) in Figure 3). As part of restart, socket connections are released with the **mssd** of s and are established with the **mssd** of s'.

4.1 Checkpointing

For checkpointing, we are using an implementation available in the public domain. To checkpoint the state of an MH, **msim** sends a signal to MH. The MH executes a signal handler to checkpoint its state. First, the data region of the process is saved. Next a *setjmp()* call is made to save the register contents into a file. The **msim** process at the destination MSS uses the checkpointed data to restart the process at the destination. Currently, we do not support file operations.

The files associated with the MH are not moved along with the MH. At present, we assume that the files in a remote machine are accessible using NFS automount (which is the case in our testbed environment). This simple option is chosen to reduce the time taken to simulate mobility.

In order to support file operations, we are planning to implement a scheme similar to that of condor [8]. The file position of all open files will be saved, and the buffers will be flushed. Since we checkpoint the state of an MH (and flush the buffers) only when the MH has to migrate, we can support combinations of read and write operations. (In condor, only idempotent file operations are supported.)

5 Handoff

After MH h is restarted by **msim** m', the following steps take place as part of a handoff procedure executed between the MSS s and the new MSS s' within the MCE. The handoff procedure described below is the same for both the actual mobile environment and the simulation testbed. In case of actual mobile environment, handoff is triggered by reception of *beacon* by MH h from MSS s'. In case of simulation, we simulate reception of beacon from MSS s' during process restart. This triggers the handoff between s and s'.

- MH h registers with MSS s' by sending a message *register*(h, s) ((4) in Figure 3). MSS s' then sends a message to the home MSS of MH h to request verification and to update the current location of h to be MSS s'.
- MSS s' sends a *deregister*(h) message to s ((5) in Figure 3).
- MSS s, on receiving deregister message, removes MH h from its list of local MHs and sets the forwarding pointer for MH h to MSS s'.
- MSS s then transfers the state of MH h stored in s (if any) to MSS s' ((6) in Figure 3). MSS s also transfers any undelivered messages buffered for MH h to MSS s'. If s receives any messages for h, they will be forwarded to MSS s'.
- MSS s' then sends a registration accepted message to h.

6 Features

The simulation testbed provides a set of features, such as *search*(h) and *deliver*(m, h), that could be used while developing, evaluating or testing new applications. These features are outlined in this section. MCE provides these as default features and the implementation details are provided in Section 8.

search(h) returns the current location of the mobile host h. The current location of h is the MSS s' in whose cell h currently resides in. Let *search*(h) be invoked at MSS s. MSS s then sends a message to the home MSS of h enquiring the current location of MH h. The home MSS, on receiving the enquiry, checks to see if the MH has been registered with any MSS by checking its database. If h is registered with MSS s_{curr}, then the location enquiry message is forwarded to s_{curr}. When s_{curr} receives the message, it verifies that mobile host h is present in its cell and sends a reply to the originator of the *search*(h) operation. If h is not present in s_{curr} and if s_{curr} has a forwarding address, say s_{next} for h, then the location enquiry message is forwarded to s_{next}. In case there is no forwarding address for h, a "not found" message is sent as a reply.

deliver(m, h) delivers message m to MH h using a routing strategy similar to the scheme designed by IETF [1]. Every mobile host has a home MSS. Whenever a mobile host registers with an MSS, it will inform the home machine of its current location. *deliver*(m, h) will send message m to h's home MSS, which will then forward it to the current MSS of h.

7 MCE Platform Design and Implementation Details

Our MCE platform is currently implemented on Pentium PCs running the Linux operating system (version 1.0.8) and on Sun workstations running SunOS 4.1.3. In our initial setup, we do not have true mobile hosts. As stated earlier, mobile hosts are simulated by processes running on the MSS itself. This will be useful to test various protocols and algorithms for the MCE platform. In our initial version, we restrict migration of an MH from MSS s to MSS s' only if s and s' have the same hardware and OS. This is due to the fact that we do process migration by checkpointing and transferring the process to the new host. Our scheme for checkpointing (as explained earlier in section 4.1) is hardware/OS specific which imposes this restriction. This is a limitation only on the simulation testbed part of MCE, since in a real mobile network, mobile hosts migrate on their own to a new MSS and MCE does not have to simulate the mobility aspect.

The MSS server process (**mssd**), which supports the MCE platform, runs in the background as a daemon process. The **mssd** server will be a hybrid server with certain services like *date and time* provided in an iterative fashion and others like *file transfer* provided in a concurrent fashion. The nature of service will dictate whether the server will do it iteratively or concurrently.

The **mssd** process handles all the MCE-related messages going through the MSS. When an MSS boots, the **mssd** process is initiated. The process then sets up the required communication interfaces using sockets to handle message traffic within the MCE. We have three distinct interfaces to handle:

- MSS to MSS static network-based communications between their respective **mssd** daemons.
- messages between an MH and the **mssd** process of its current MSS.
- messages between processes (other than **mssd**) in an MSS and its **mssd** process (intra-MSS MCE communications). This includes supporting **msim** communications.

Once the sockets are created and bound to addresses, we set up the **mssd** process to handle the asynchronous messages arriving on any of the above mentioned interfaces. Another step in the initialization phase is to setup tables needed to verify/authenticate mobile host data. In our initial version, we have two tables as follows:

- **HMSS** database table to keep track of mobile hosts whose home MSS is the local host.
- **VMSS** table to keep track of all mobile hosts currently getting serviced by the MSS. This includes all the mobile hosts that have been registered with the local MSS regardless of their home MSSs.

In order to support the development of mobile host processes, we have developed a *MCE Application Programming Interface* (**MCE-API**). The purpose of this interface is to provide interface routines that will make the underlying issues of mobility etc., transparent to the application developer so that the developer does not require an in-depth understanding of 'how' the MCE works. The knowledge of 'what' the MCE provides is sufficient. The MCE-API also provides a way to ensure that the applications need not be rewritten or modified irrespective of whether the process is being simulated on the MSS itself or is actually run on a mobile host with a wireless communication interface. The MCE-API provides interfaces to handle response to beacon messages for registration, to setup connections, to send and receive messages and to terminate connections. The software architecture and message flow paths are shown in Figure 4.

8 Current Status Report

At present, we have MCE simulation testbed working in our lab with a default routing scheme, default search facility, default registration and default deregistration features. We describe these features in detail and state the current development status along with immediate and future plans. We are planning to add support to let users add similar services without having to modify MCE source code. We view this as a subscription based service and the motivation behind this idea is to be able to use MCE as an evaluation and/or comparison platform to evaluate different algorithms. For example, if there are two routing algorithms to be compared, one can run test cases on both algorithms using the same benchmark applications and compare the results. MCE keeps environment used on both algorithms the same which helps the user get more dependable results.

The following is a list of terms used in the rest of this section.

- h - A mobile host.
- s_{new} - The MSS of the cell into which h has just moved into during cell to cell migration.
- s_{prev} - The MSS of the cell from where h migrated.
- s_{curr} - The MSS of the cell where h currently resides.
- s_{home} - The home MSS of h.

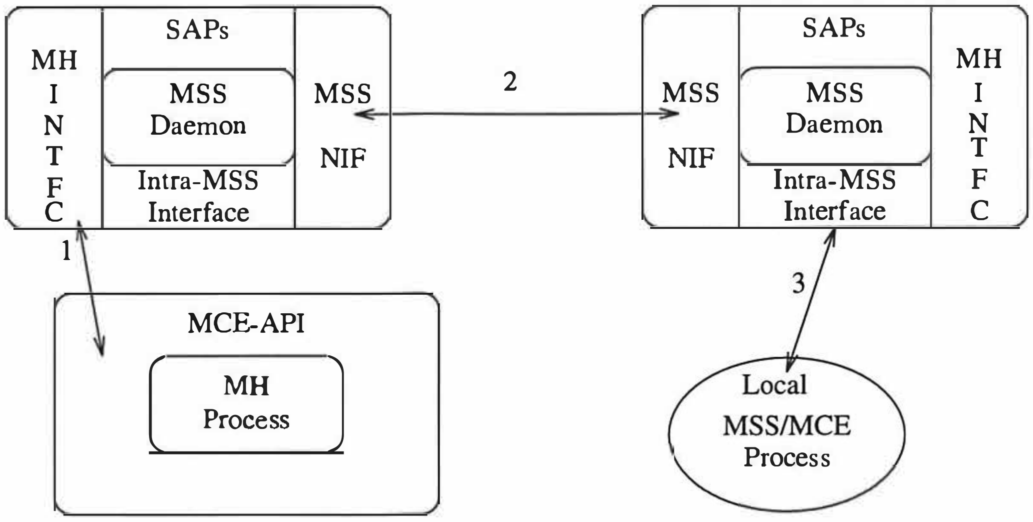
8.1 Simulation Testbed Setup

Figure 5 shows the backbone local area network on which MCE is being developed and tested. Files in any of the machines may be accessed from any other machine using *automount* facility. Each machine is represented by a node in the diagram and the node contains the name[1] and the operating system of the machine. As can be seen, the MCE simulation platform is running on different flavors of the UNIX operating system. The checkpointing scheme that we use is also OS/hardware specific. But the evolution of MCE is being planned to be independent of any operating system and/or hardware specific idiosyncracies as much as possible. The PDAs, notebooks etc. are mostly available on the DOS/Windows operating environments and we are planning to tune MCE-API to be usable on these platforms as well.

8.2 Limitations

We simulate a mobile host by a single process at this time. This poses a limitation on the use of MCE as this restricts the ability to test multiple concurrent applications on a single mobile host. However, the current version of MCE may be effectively used to test applications that are independent of other processes on the mobile host. This implies that we simulate these independent mobile host processes as if they were running on different mobile hosts. Note that this still does not impose or introduce application code that is specific to the simulation environment. But this does not aid in the development of inter-dependent, concurrent mobile host applications. We are working on extending the MCE

[1]The internet address of the machines may be derived by just appending "*.utdallas.edu*" to the names shown



1. Mobile Host <-> MSS Daemon Interface (MH INTFC)
2. MSS Daemon <-> MSS Daemon Network Interface (MSS NIF)
3. MSS Local Process <-> MSS Daemon Interface (Intra-MSS Interface)

MCE-API - MCE Application Programming Interface
SAPs - Service Access Points

Figure 4: MCE Software Architecture

simulation facility to support development of inter-dependent concurrent processes.

The checkpointing method used is another limitation within MCE simulation testbed. We are exploring methods to do checkpointing and recovery in a machine independent way to overcome the limitation imposed by our current method.

8.3 Utilities

Each mobile host must have subscribed for service with its home MSS. MCE provides a **subscribe** utility that takes subscription data from a user and adds the data to the HMSS database. The HMSS database stores the mobile host's unique network identifier, subscribed services, privileges and last known location for the mobile host. In the current implementation, we use a 2-tuple, [home MSS network address, mobile host identifier] to form the unique network-wide mobile host identifier. Whenever a mobile host registers for service with any MSS, the HMSS database is updated to reflect the current location of the mobile host. There are no service subscription data stored in HMSS database now but as MCE evolves to support various subscribable services like *ftp*, *email* etc., the subscription tool will evolve to support them. MCE does not have a browser tool to browse the HMSS database and we are working on adding such a tool in the near future. Each MSS has an in-memory database called **VMSS**, to keep track of the mobile hosts that are getting serviced by the MSS at any given time.

8.4 Registration

MCE requires a mobile host to register with the local MSS prior to requesting or receiving any service. MCE provides a default registration service for mobile hosts to register with their local MSS. Mobile hosts may request registration services with the local MSS under (at least) two different circumstances and both of these are explained below.

- MH h is powered up and hence does not have an MSS that is currently providing service. In this scenario, when h receives a *beacon* from the MSS of the cell it currently resides in, say s_{new}, h sends a registration request to s_{new}. s_{new} forwards the message to the home MSS of h,

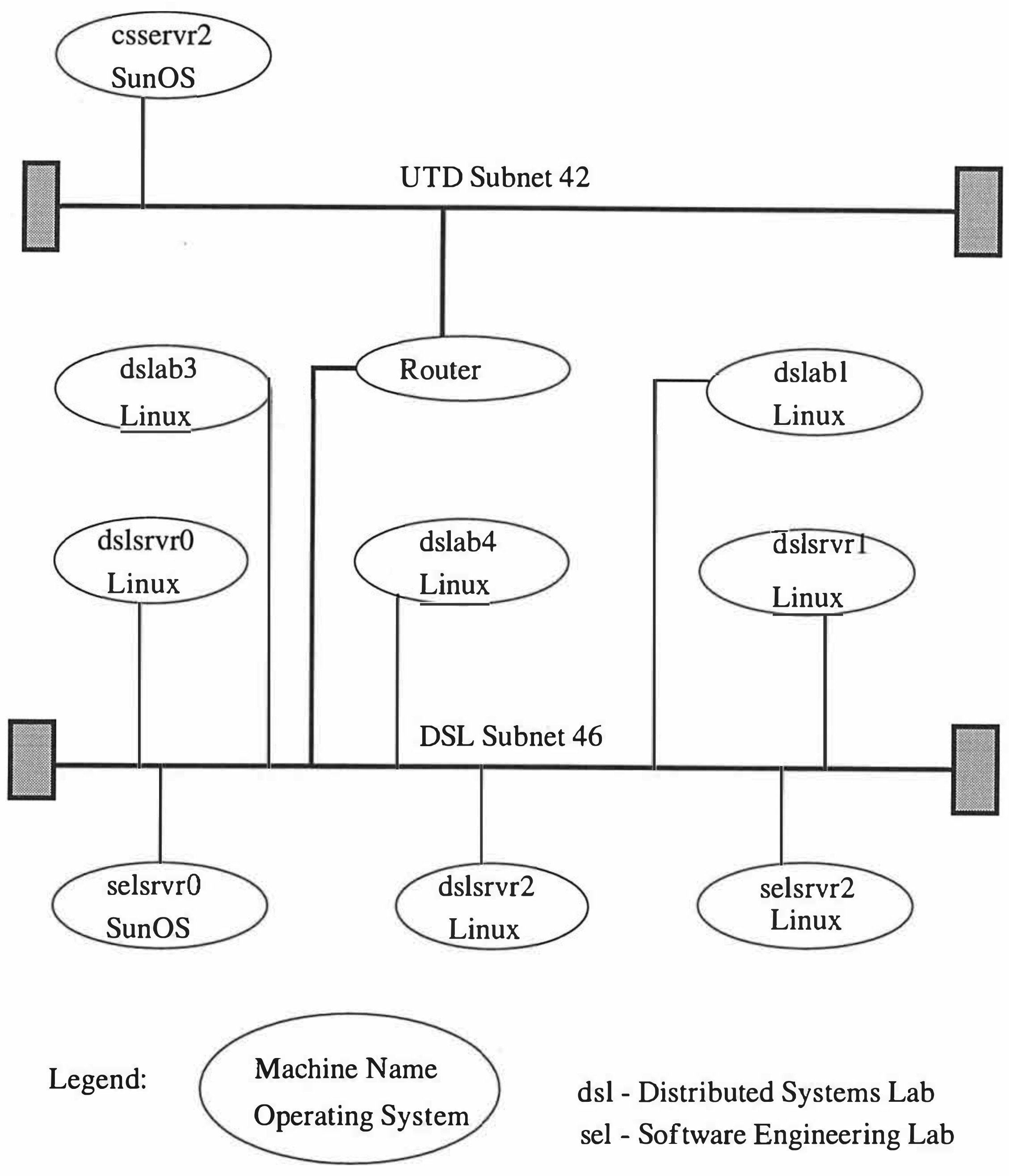


Figure 5: MCE Simulation Testbed Setup

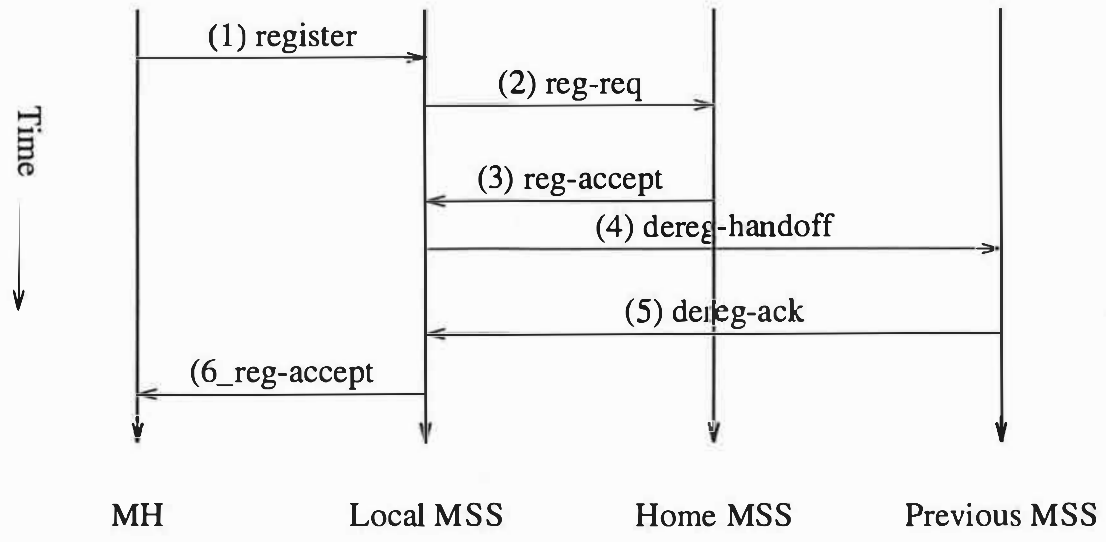


Figure 6: Mobile Host Registration Message Ladder

say s_{home}. When s_{home} receives the registration request, it validates the information and sends an *accept* or *deny* message. If s_{home} accepts the registration request, the HMSS database is updated with the location information about h. s_{new} receives this message and forwards it to h. If the registration was accepted, s_{new} also modifies the VMSS tables with data about h. Once this registration process is completed, MH h may receive messages from anywhere in the network at the new location via s_{new}.

- MH h has just moved into the cell of s_{new} and receives a *beacon* from s_{new}. The MSS address is different from what h remembers and so h sends a registration request to the new MSS s_{new}. The sequence is the same as the previous case till s_{new} receives a reply from s_{home}. If the registration succeeds, s_{new} sends a deregistration due to handoff request to the previous MSS as remembered by h, say s_{prev}. When s_{prev} receives the deregistration due to handoff message from s_{new}, all buffered messages for h are forwarded to s_{new} and then the VMSS table entry for h in s_{prev} is deleted. s_{prev} then sends a deregistration acknowledgement to s_{new}. When s_{new} receives this acknowledgement, it updates its VMSS table for h and then forwards an accept message. If registration was denied, no deregistration request is sent by s_{new}.

Message ladder depicting the second registration scenario is shown in Figure 6. The default registration algorithm used by MCE is presented in Appendix A.1.

8.5 Deregistration

MCE uses a deregistration message to keep the location tracking of mobile hosts up-to-date in VMSS and HMSS databases. In case of deregistration also, there are two scenarios that are supported by MCE.

- Mobile host h is going to be shut down. This may result in the mobile host sending a deregistration with disconnect as the reason. When s_{curr}, the current MSS of h, receives it, s_{curr} will forward the request to s_{home}. Then s_{curr} will clear the data about h from its VMSS table. When s_{home} receives the deregistration *disconnect* message, it updates its HMSS database to reflect h's disconnected state from the network.

- The disconnect message is sent as part of the handoff sequence as described in registration earlier. The disconnect message sent by s_{new} will indicate handoff as the reason. When s_{prev} receives this message, it forwards all the buffered messages for h to s_{new}, then sends a deregistration acknowledgement and clears VMSS table of h's data.

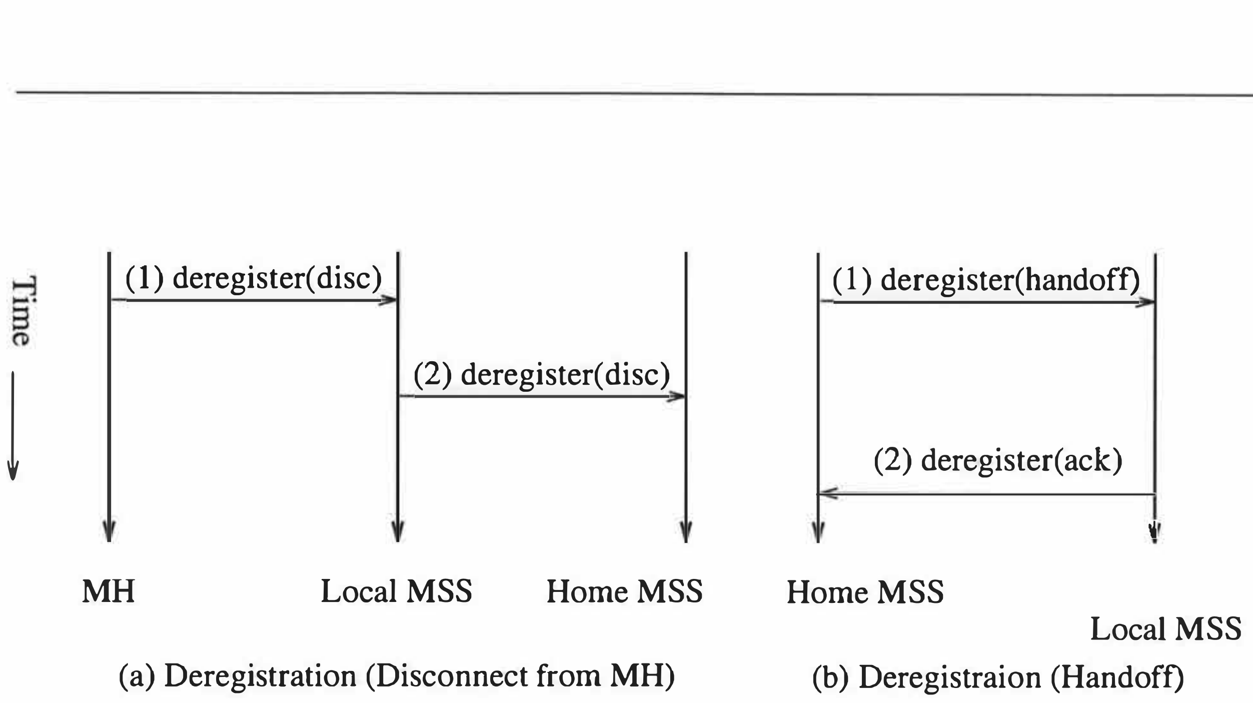


Figure 7: Mobile Host Deregistration Message Ladders

Message ladders for these deregistration scenarios are presented in Figure 7. The default deregistration algorithm used by MCE is presented in Appendix A.2.

8.6 Routing Scheme

MCE provides a message routing mechanism, which is similar to IETF [1]. MCE assumes that message delivery between mobile hosts is done using their respective mobile support stations. This assumption not only helps in simulation but also in providing other services like tolerating MSS failures [2]. The default scheme provided for routing within MCE is built using datagram sockets on UDP/IP protocols above the transport layer. This may be modified to be part of IP layer in the future. MCE's default routing scheme is illustrated in Figure 8. When a mobile host h_{src} sends a message m_1 to another mobile host h_{dest}, the message m_1 is first received by the current MSS of h_{src}. The message is then forwarded to the home MSS of h_{dest}. Home MSS of h_{dest} looks up h_{dest}'s location in its HMSS database and forwards m_1 to the current MSS of h_{dest}. When the current MSS of h_{dest} receives m_1, it looks up its VMSS database for information on h_{dest} and delivers the message to h_{dest}. The routing scheme is not reliable and assumes that applications have mechanisms to ensure reliable message delivery. We are planning to add reliable message delivery options in the near future. The routing algorithm executed at every MSS is described in Appendix A.3.

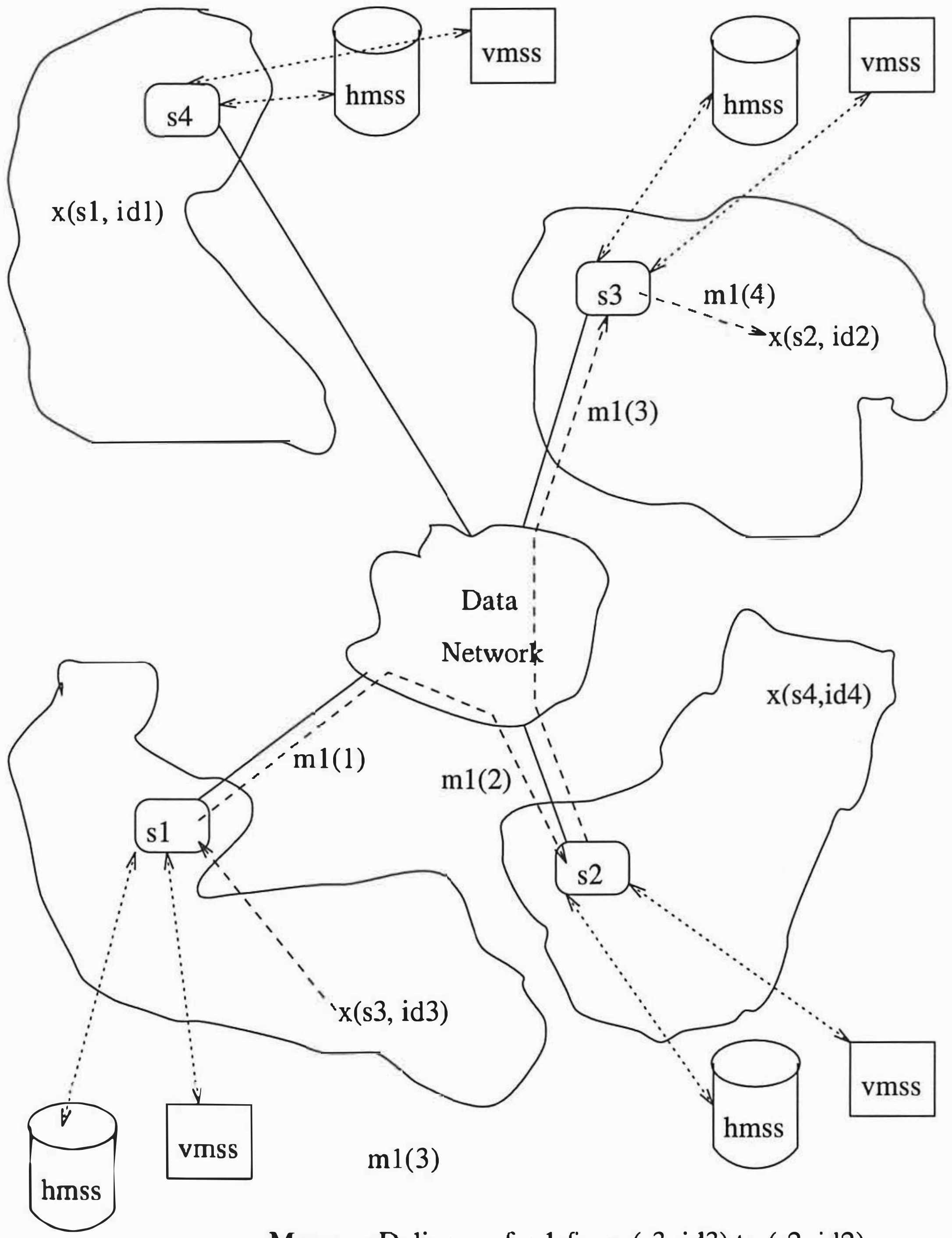
8.7 Mobile Host Search Facility

The *search*(h) feature as described earlier in Section 6 is provided by the current version of MCE. The implementation uses the default routing scheme. The location enquiry message is forwarded to the MSS where h resides. Instead of delivering the message to h, the local MSS returns a message giving the current location information to the originator of the *search*(h) request.

9 Future Work

We plan to enhance MCE so as to make it a complete simulation testbed as well as a platform to support a mobile network environment. Currently we are planning to add support to provide subscribable services like *ftp*, *email*, *talk* etc. We are working on schemes to let the user subscribe for a service either on an iterative or on a concurrent basis. We plan to add fault tolerance [2] and study the performance. In addition, we are planning to provide support to do performance comparison and analysis of routing schemes. Support for concurrent mobile host processes is also being planned for the near future.

An *ftp site* for accessing MCE software and documents will be setup in the very near future. Once



Message Delivery of m1 from (s3, id3) to (s2, id2)

Figure 8: MCE Default Message Routing Scheme

setup, the access will be provided through *anonymous ftp* to *ftp.utdallas.edu*. The proposed directory is *pub/cs/venky/MCE* and a README file in that directory will explain the setup procedures for MCE.

Acknowledgements: We would like to thank the anonymous referees for their insightful comments and suggestions.

References

[1] draft-ietf-mobileip-protocol-07.txt. In *INTERNET DRAFT*, C. Perkins, Ed. Internet Engineering Task Force, 1994.

[2] ALAGAR, S., RAJAGOPALAN, R., AND VENKATESAN, S. Tolerating mobile support station failures. Computer science technical report, The University of Texas at Dallas, November 1993.

[3] ALAGAR, S., AND VENKATESAN, S. Causal ordering in distributed mobile systems. In *Workshop on Mobile Computing Systems and Applications* (1994).

[4] BADRINATH, B., AND ACHARYA, A. Delivering multicast messages in networks with mobile hosts. In *Proceedings of the 13th International Conference on Distributed Computing Systems* (1993), IEEE, pp. 292–299.

[5] DIEHL, N., AND HELD, A. A system platform for mobile computing applications. In *MobiData Workshop* (1994), Rutgers University.

[6] IOANNIDIS, J., DUCHAMP, D., AND MAGUIRE, G. IP-based protocols for mobile internetworking. In *Proceedings of ACM SIGCOMM Symposium on Communication Architecture and Protocols* (1991), pp. 235–245.

[7] JOHNSON, D. Scalable and robust internetwork routing for mobile hosts. In *Proceedings of the 14th International Conference on Distributed Computing Systems* (1994), IEEE.

[8] LITZKOW, M., AND SOLOMON, M. Supporting checkpointing and process migration outside the unix kernel. In *Proceedings of Usenix Winter Conference* (1992).

A MCE Default Algorithms

A.1 MCE Registration Algorithm

```
if the message is from a local mobile host
    if home MSS of the mobile host == local MSS
        update current location information on HMSS databse;
        update the VMSS table for mobile host interface data;
        if the registration is part of handoff
            send deregister message to previous MSS;
        else
            send regsiter accept message to mobile host;
    else if home MSS is not the local MSS
        forward the register message to the hmss MSS of the mobile host;
else if the message was received on the MSS network interface
    if the message is a registration request message
        if local MSS is not the home MSS of the mobile host
            ignore the message and return;
        retrieve the profile of the mobile host from HMSS database;
        verify the mobile host request and validate;
        if registration is valid request
            update current location information on HMSS database;
            send registration accepted message to the new MSS of the mobile host;
        else
            send a registration denied message to the new MSS of the mobile host;
    else if the message is a registration accepted message
        if registration is due to handoff
            send a deregister message to the previous MSS of the mobile host;
        else
            send registration accepted message to the mobile host;
    else if the message is a registration denied message
        send registration denied message to the mobile host;
        delete any current information on the mobile host;
```

A.2 MCE Deregistration Algorithm

```
if the deregister message is from a local mobile host
    update the VMSS table for mobile host interface data;
    if the home MSS of the mobile host is local MSS
        update current location information on HMSS databse;
    else if home MSS is not the local MSS
        forward the deregister message to the home MSS of the mobile host;
else if the message was received on the MSS network interface
    if the message is handoff deregistration message
        if local MSS is not the home MSS of the mobile host
            ignore the message and return;
        forward all the buffered data for the mobile host to the source MSS of the message;
        cleanout mobile host data in VMSS table;
        send deregistration accepted message to the source MSS of the message;
    else if the message is a disconnect deregistration message
        cleanout mobile host data in VMSS table;
    if this MSS is the home MSS of the mobile host that is getting deregistered
```

```
        update the HMSS database to reflect the disconnected state of the mobile host;
    else if the message is a deregistration acknowledgement message
        /* this is the reply to deregistration message */
        /* to the previous MSS during handoff */
        update VMSS table with the mobile host data;
```

A.3 MCE Routing Algorithm

```
if message was received on the mobile host (wireless) interface:
    if home MSS of destination is the local MSS
        find location information of mobile host from HMSS database;
        if mobile host is in cell of home MSS
            send message to destination mobile host;
        else
            forward the message to the MSS of current location of the destination mobile host;
    else if the home MSS of destination is different
        forward the message to the home MSS of the destination;
else if message was received on the MSS network interface:
    if home MSS of destination is the local MSS
        find location information of mobile host from HMSS database;
        if mobile host is in cell of home MSS
            send message to destination mobile host;
        else
            forward the message to the MSS of current location of the destination mobile host;
    else /* not home MSS of destination mobile host */
        if destination mobile host is registered with local MSS
            send message to destination mobile host;
        else if destination host has buffering setup
            buffer the message to be forwarded later;
        else
            message gets dropped;
```

A NETWORK EMULATOR TO SUPPORT THE DEVELOPMENT OF ADAPTIVE APPLICATIONS

Nigel Davies, Gordon S. Blair, Keith Cheverst and Adrian Friday

Distributed Multimedia Research Group,
Department of Computing,
Lancaster University,
Bailrigg,
Lancaster,
LA1 4YR,
U.K.

telephone: +44 (0)524 65201
e-mail: nigel, gordon, kc, adrian@comp.lancs.ac.uk

Abstract

Mobile applications must operate in environments in which the network connectivity, input/output devices, power and contextual information available to them may all vary. Applications which react to changes in these parameters in order to ensure continuing service to the user are termed *adaptive applications* and have recently emerged as an area of intense research activity. In this paper we describe the design and implementation of a network emulator which facilitates research in this field by allowing applications to be exposed to user controlled fluctuations in network service. The emulator can be used with any application which uses UDP and requires only minimal changes to the application or, it may be used with applications written using the ANSAware distributed systems platform in which case no changes are necessary to the application. The design and implementation of the emulator are described in this paper as our experiences of using the emulator to model three distinct types of wireless network: GSM, an analogue cellular service and a simple shared radio channel. The source code for the emulator is freely available and instructions on obtaining the code are also included.

1. Introduction

Mobile computing environments are characterised by *variation*. In particular, during the execution-time of a mobile application the network connectivity, input/output devices, power and contextual information available to the application may all vary [Davies,94], [Duchamp,92], [Schilit,94]. In our research at Lancaster we are interested in developing applications and system services which are able to cope with wide fluctuations in these parameters (termed adaptive applications [Katz,94]) and in particular with fluctuations in the first of these parameters, i.e. network connectivity. This work has been motivated by two beliefs: firstly, that fluctuations in network connectivity are unavoidable in a mobile environment and secondly, that such fluctuations should become an accepted part of an application's operation and not be treated as an error or temporary 'glitch'.

The first of these statements is clearly true if we assume that mobile computers will have multiple network interfaces [Hager,93] and that users will be able to dynamically switch between networks (for example switching between a wired network and a local area wireless network when un-docking a portable PC). Determining the feasibility of the second of these statements requires further research. However, to conduct this research requires an environment in which the level of network service available to a mobile machine can be varied. This paper reports on the development of such an environment based not on hardware but on a software network emulator which allows us to conduct research into network variance without investing in multiple network infrastructures.

Section 2 of this paper describes the overall design and configuration of the network emulator. Details are given of how the emulator may be configured to emulate a number of different networks with examples based on three wide-area wireless networks with which we have practical experience: GSM, a U.K. analogue cellular service and an analogue private mobile radio (PMR) system. Section 3 then briefly describes a graphical user interface to the emulator which enables users to control and visualise the flow of information between mobile computers. Section 4 presents details of the modifications necessary to client applications to enable them to exploit the emulator. Particular emphasis is placed on the use of applications written using the ANSAware distributed systems platform which we have modified to operate with the emulator. Section 5 then presents an analysis of the performance of the emulator and highlights the relationship between the network bandwidth to be emulated and the average size of packets sent to the emulator. Finally, section 6 contains some concluding remarks.

2. Design and Implementation

2.1. Emulator Design

The network emulator is designed to provide an approximate emulation of low-speed networks using standard hardware and systems software. It should be stressed that as researchers we are more concerned with variance in network connectivity than precise simulations of network characteristics and hence the accuracy of the emulator is not considered to be of critical importance (for example, we use standard UNIX timing facilities). The basic approach used by the emulator is to intercept UDP packets travelling between sources and sinks and to introduce a delay similar to that which would be incurred if the packets were transmitted over a slower network. The emulator mimics the workings of a slow speed network and so delays are related to (for example) network load and error rates. The most controversial feature of the emulator's design is that it is structured as a single, central point through which all messages are routed and at which point network delays are introduced. Thus for each node in the network to be emulated the emulator maintains a queue of packets waiting to be transmitted. This use of a central point for the emulation is in contrast to systems such as Ingham's Delayline network emulator [Ingham,94] in which processing is carried out at both the sender and recipient of messages with delays being implemented at the receiver's end.

While implementing the emulator as a central point clearly creates a bottleneck in the system there are two key advantages to be gained from this approach. Firstly, the emulator is able to adjust the network characteristics experienced by applications based on load. Hence, for example, if the network to be emulated has a simple shared transmission medium the emulator itself can detect potential packet collisions and discard the appropriate packets. The second advantage is that the semantics of sockets are automatically preserved by the emulator: the sender always believes that packets have been sent properly since they always appear to reach their destination (in practice of course they have only reached the emulator) and the receiver receives messages in the order in which they would arrive in a real network (in contrast to the Delayline system in which packets may arrive in the wrong order since the delay is introduced at the receiver side of the communication).

However, there are a number of disadvantages in structuring the network emulator as a central process. In particular, the design makes the following assumptions:-

- The time the emulator takes to process a packet is negligible compared to the delay incurred during transmission over a slow network.
- The time taken to transmit a UDP packet over the high speed network is negligible compared to the delay incurred during transmission over a slow network.
- The number of nodes that are to be interconnected via the emulator is small (i.e. less than sixteen) and only a subset of these are transmitting at any one time.

Clearly, as the size of UDP packets decrease or the speed of the network being emulated increases then the first two of these assumptions introduce increasing inaccuracies. However, in practice we have found that these assumptions are valid for the type of experimental work we wish to carry out (see section 5).

2.2. Emulator Configuration

The network emulator can be configured in two distinct ways. Firstly, new types of network may be introduced, e.g. a connection oriented cellular service or a connectionless shared medium network. This requires modification to one of the emulator's source files and re-compilation. In more detail, the user must supply a function called new_network_name_send (senderNodeId, dataPacket) which is called by the emulator every time a packet is to be sent via the new network. Within this function the user must implement any delays which are associated with attempting to send packets on the network. For example, if the network has a high turn-around time which occurs when the node switches from receiving to transmitting information this can be modelled with the new_network_name_send function. Error characteristics can also be specified for the network or the occurrence of errors may be modelled as part of the throughput specified. Once a new network has been introduced its behaviour can be tailored during run-time using configuration files. A typical configuration file is shown in figure 1.

Line 1 of the configuration file denotes the type of the network to emulate - in this case a raw radio channel. Line 2 specifies the number of nodes that are connected to the network (in this case 3) and lines 3 to 5 provide information about each node. Specifically, for each node its name, maximum buffer size and internet address must be specified. The buffer size is used to

```
1     raw                                  <network to be emulated>
2     3                                    <no. of sources/sinks of data>
3     0 1044000 148 88 16 27 columbine     <source no. 0>
4     1 1044000 148 88 16 25 sinbad        < source no 1>
5     2 1044000 148 88 32 2 edc2           <source no. 2>
6     0 0 1 1200 3c                        < channel characteristics 0-1>
7     1 0 2 1200 3c                        < channel characteristics 0-2>
8     2 1 0 1200 33                        < channel characteristics 1-0>
9     3 1 2 1200 33                        < channel characteristics 1-2>
10    4 2 0 1200 0f                        < channel characteristics 2-0>
11    5 2 1 1200 0f                        < channel characteristics 2-1>
```

Figure 1 : A Typical Configuration File

prevent applications from running ahead of the network; once a node's buffer is full all subsequent send requests will be blocked.

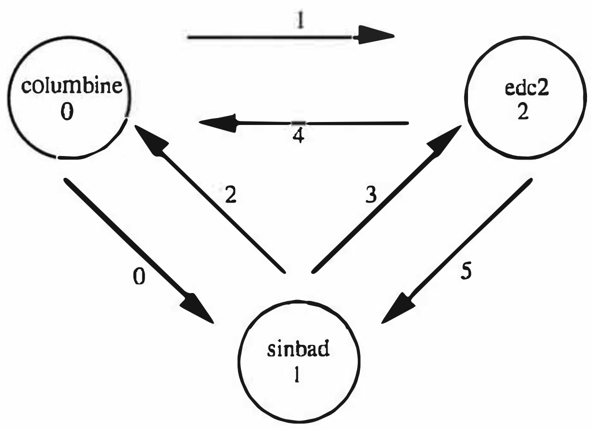


Figure 2 : Emulator Channel Connection Diagram

For the purposes of the emulator the network to be emulated must be visualised as a series of uni-directional channels interconnecting each of the nodes (see figure 2). The characteristics of these channels are specified in lines 6-11. For example, line 6 of the configuration file specifies the characteristics of channel 0, i.e. the channel between nodes 0 and 1. The characteristics are that the channel has a throughput of 1200 bps and that messages transmitted on this channel collide with messages transmitted simultaneously on any of the *other* channels except channel 1, i.e. the other outgoing channel from this node. This is expressed using a bit map mask with a bit being set denoting that collisions occur with messages on the corresponding channel (channel 0 being the least significant bit).

The network emulator supports dynamic updates to the configuration file during operation so that the effect of changing the quality-of-service of a network can be easily demonstrated. For example, by simply setting the throughput of a given node's output channels to zero we can emulate disconnection. It should be noted however that radical changes to the network configuration may result in those packets currently being queued at the emulator being delayed longer than expected during the reconfiguration. This is because when the emulator is re-configured it re-calculates the dispatch time of all the waiting packets without taking into account any time the packets have already been delayed. Thus re-configurations involving small changes to the throughput of slow-speed networks are most susceptible to this problem (particularly if the packets being queued are relatively large). Re-configurations involving the addition or removal of nodes are supported: the emulator simply prints a warning if a previously supported node is no-longer supported in the new configuration file.

2.3. Example Configurations

We have used the emulator to emulate three types of network: GSM, a U.K. analogue cellular service and a simple shared radio channel. In the case of GSM the network appears to have fairly dependable characteristics with an average call set up time of 3 seconds and a corrected throughput of 9600 bits/sec. In the case of the analogue cellular service (using Motorola Cellect modems and MicroTac II handsets) the call set up time is substantially longer, taking about 20 seconds for the connection to be fully established. Once the connection has been established we are able to get a corrected throughput of around 3700 bits/sec on a theoretical 4800 link. Setting the modems to run at higher speeds typically gives us a lower corrected throughput due to the number of retransmissions necessary to compensate for the high error rate.

The emulator configuration for these two networks is very similar. Two new send routines were required (gsm_send () and analogue_send ()) which introduced the appropriate connection delay for each network. For both networks packets sent by a node to a new destination causes the emulator to simulate the disconnection of the node from its previous destination and connection to the new destination node. The only difference between the gsm_send and analogue_send routines is the length of delay they introduce to emulate call connection. The configuration files for these networks are both straightforward with the collisions flags being set to no collisions and the throughput being set at 9600 and 3700 for the GSM and analogue networks respectively.

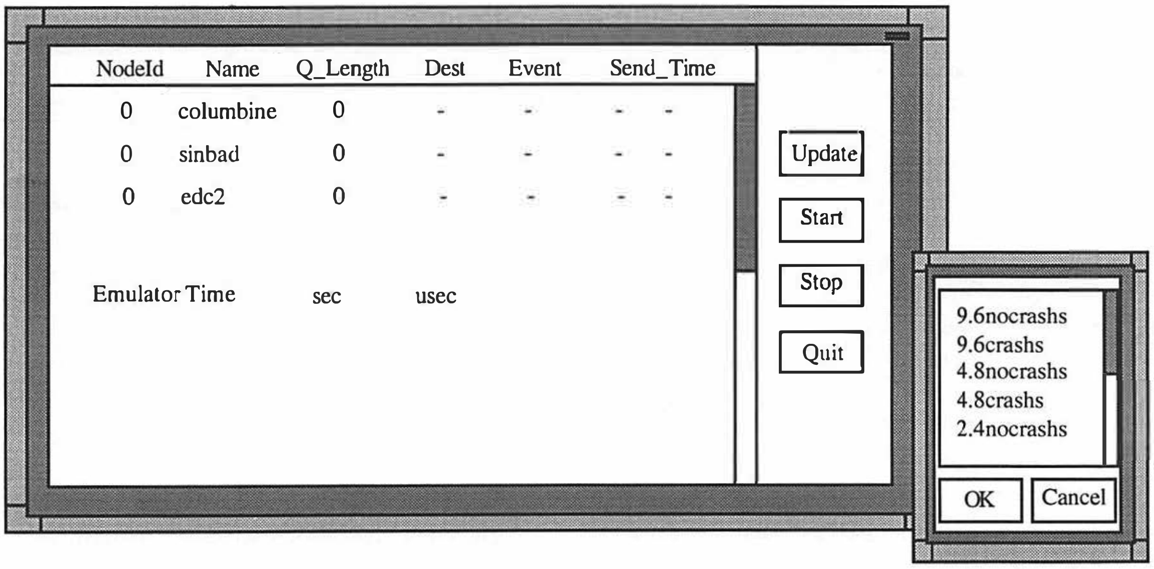
```
raw_send ( int : sourceld, dataPacket *pkt)
{
    mapAddressToNodeld (dataPkt->
            destinationAddress, &sinkId);
    if (emptyQ(sourceld) {
        addToTxQ (sourceld, pkt);
        obtainChannelCharacteristics(sourceld,
            sinkId, &characteristics);
        /* insert any additional delays/errors here */
        calculateDispatchTime(pkt->length,
        characteristics->bandwidth);
        resetCollisionBits (characteristics);
    }
    else
        addToTxQ (sourceld, pkt);
}
```

Figure 3 : Pseudo Code for Emulator Send Routine for a Raw Radio Channel

The emulation of the raw radio channel has a much more straightforward send routine which introduces no additional delays (see figure 3). However, the configuration file for this type of network is more complex. In particular, the collision flags must be set such that data on any channel collides with data on any other channel. For the purposes of our work we have used a throughput of 1200 bits/sec to emulate the characteristics of a simple analogue private mobile radio (PMR) system. An example configuration file for this type of network is given in figure 2.

3. A Graphical Interface to the Network Emulator

At an early stage in the network emulator's development it was realised that a graphical front-end to the emulator could be used to enhance demonstrations of adaptive applications. The interface we have developed allows users to both view and control the operation of the emulator. During normal use the interface displays for each node the number of packets waiting to be dispatched, the last action that occurred with respect to that node (e.g. packet arrived, packet dispatched etc.) and for the packet at the head of the node's queue its destination, size and dispatch time. Hence, if we have a fast sender connected to its intended destination by a slow network the queue size for the sending node will build up steadily and we will see many more packet arrival events than packet departures. The interface to the emulator also allows users to control the emulator by dynamically changing the configuration file it uses. In this way we can, for example, show the effect on applications of gradually reducing the throughput available.



NodeId	Name	Q_Length	Dest	Event	Send_Time
0	columbine	0	-	-	- -
0	sinbad	0	-	-	- -
0	edc2	0	-	-	- -

Emulator Time sec usec

9.6nocrashs
9.6crashs
4.8nocrashs
4.8crashs
2.4nocrashs

Figure 4 : The Network Emulator Controller

The interface is implemented as an entirely separate process which communicates with the emulator using sockets. This communication takes the form of well-defined packets sent from the emulator whenever a relevant event occurs (see figure 5). Control packets to, for example, force the emulator to update its configuration file can be sent from the monitor to the emulator.

```
struct monitorPacket {
    char header [2];    /* identifies emulator pkt */
    int type;           /* type of pkt */
    struct timeval
        time_now;       /* emulator's clock time */
    int nodeId;         /* node to which msg relates */
    int qLength;        /* length of q for this node */
    int event;          /* event which has occurred */
    int size;           /* size of data pkt involved */
    int dest;           /* destination nodeId */
    struct timeval
        event_time;     /* time relating to the event.
                           Can be different for each
                           event */
};

/* pkt has been discarded */
#define EVENT_PKT_DISCARD           0
/* pkt has been sent */
#define EVENT_PKT_SENT              1
/* pkt collided (and discarded) */
#define EVENT_PKT_COLLIDED          2
/* pkt has arrived at node */
#define EVENT_PKT_ARRIVED           3
/* pkt has been scheduled for tx */
#define EVENT_PKT_SCHEDULED         4
```

Figure 5 : Packet Format For Communications Between the Emulator and its Interface

Implementing the emulator's user interface as a separate process has the two distinct advantages. Firstly, we can run the interface on a separate machine and thus could implement processor intensive graphics monitoring tools without affecting the performance of the emulator. Secondly, we can have a number of different interfaces implemented to illustrate and control different aspects of the emulator.

4. Emulator Client Code

4.1. Standard Distributed Applications

We use the emulator with two types of distributed application. The first are standard distributed applications which communicate using UDP. In order that these can use the emulator they must use new versions of the sendto and recvfrom system calls. These are currently implemented as new functions emulator_sendto and emulator_recvfrom which form wrappers around the standard calls in order to add and remove additional header information required by the emulator. Applications must at present be re-compiled to use these new functions. However, it would be a relatively straightforward task to compile these functions as a library which could be dynamically linked with existing applications to allow them to transparently use the emulator. The format of the packet headers used by emulator_sendto and emulator_recvfrom is shown in figure 6.

```
struct dataPacket {
   char    header [2];              /* identifies em. pkt */
   int     type;                    /* type of pkt */
   struct  sockaddr_in toAddr;      /* destination */
   struct  sockaddr_in frAddr;      /* source address */
   struct  sockaddr_in ackAddr;     /* address to ack.
                                       transmission */
   int     bufLen;                  /* length of user data */
   char    *buf;                    /* user data */
};
```

Figure 6 : Structure of an Emulator Data Packet

The header field identifies the packet as being associated with the emulator. It is used by the emulator to check that it is receiving valid packets and by the emulator_recvfrom function to determine whether or not to strip off the header before passing the buffer up to the application. The type field is used to distinguish between data and control packets. Data packets are those which are passed to the emulator for subsequent dispatch to a destination mode. Control packets are used to control the emulator's behaviour and typically originate from the emulator's user interface. In addition data packets can be flagged as those requiring an acknowledgement that the packet has been queued for transmission, those that require an acknowledgement of transmission and those which require no acknowledgement at all. This allows us to implement synchronous emulator_sendto routines for those applications which would otherwise 'run-away' or cause congestion when operating over a low-speed link. If the packet requires an acknowledgement the emulator sends this to the address specified in the ackAddr field.

The first two address fields are used by the emulator to ensure that the packet is transmitted to the appropriate final destination and, at the destination, to ensure that the application believes that the packet originates from the initial source rather than from the emulator. The ack_address is used for flow control between the emulator and the source application as described above.

4.2. ANSAware Applications

The second type of application we have used with the emulator are those based on the ANSAware distributed systems platform [APM,89]. This software suite is itself based on the ANSA architecture which has had a profound influence on the RM-ODP [ISO,92]. Thus, the platform tackles the problem of developing applications to operate in a heterogeneous environment. The ANSA programming model is based on a location-independent object model where all interacting entities are treated uniformly as encapsulated objects. Objects are accessed through operational interfaces which define named operations together with constraints on their invocation. Objects are made available for access by exporting interfaces to a special object known as the *trader*. An object wishing to interact with this interface must then import the interface from the trader by specifying a set of requirements in terms of a interface type and attribute values. This will be matched against the available services and a suitable candidate selected. At this stage, an implicit *binding* is created to the object supporting the interface, i.e. a communication path is established to the object. Invocation of operations can then proceed.

To provide a platform conformant with the above programming model the ANSAware suite augments a general purpose programming language (usually C) with two additional languages. The first of these is IDL (Interface Definition Language), which allows interfaces to be precisely defined in terms of operations, arguments and results. The second language, DPL (Distributed Processing Language) is embedded in a host language, such as C, and allows interactions to be specified between programs which implement the behaviour defined by these interfaces. Specifically, DPL statements allow the programmer to import and export interfaces, and to invoke operations in those interfaces (see figure 7).

! {stack} <- traderRef$Import ("Stack", "*context*", "*properties*")

! {result}<-stack$Push (value)

Figure 7 : Example DPL Statements

In the engineering infrastructure, the binding necessary for invocations is provided by a remote procedure call protocol known as REX (Remote EXecution protocol) or a group execution protocol know as GEX (Group EXecution Protocol). These are layered on top of a generic transport layer interface known as a *message passing service* (MPS). A number of additional protocols may be included at both the MPS and the execution protocol levels and these may be combined in a number of different configurations. The infrastructure also supports lightweight threads within objects so that multiple concurrent invocations can be dealt with.

All the above engineering functionality is collected into a single library, and an instance of this library is linked with application code to form a *capsule*. Each capsule may implement one or more computational objects. In the UNIX operating system, a capsule corresponds to a single UNIX process. Computational objects always communicate via invocation at the conceptual level but, as may be expected, invocation between objects in the same capsule is actually implemented by straightforward procedure calls rather than by execution protocols.

We have developed a modified version of the ANSAware libraries which includes code to route packets generated as a result of object invocations via the emulator. By use of a single function call the application can optionally enable one or other of the synchronous transmission modes supported by the emulator, i.e. application is blocked until messages are queued or application is blocked until messages are transmitted. Running ANSAware applications over the emulator highlighted a number of shortcomings in the ANSAware remote procedure call protocol REX. More specifically, REX is tuned to run on a moderately loaded Ethernet and does not implement any form of congestion control. In addition, the tuning parameters are specified at compile time which makes it impossible for REX to adapt to changes in network bandwidth.

We have implemented a new remote procedure call package for ANSAware called QEX (Quality-of-service remote EXecution protocol). QEX differs from REX in that it is specifically designed to operate over a wide range of network types adapting seamlessly to changes in network quality-of-service. This is achieved by analysing sequences of messages to determine the round-trip time between client and server. These round-trip times are smoothed to eliminate network jitter (processing at the server end ensures that application delays are eliminated from the calculation) and then form the basis of tuning parameters. In particular, retry rates are calculated to avoid unnecessary network congestion while ensuring that packet losses are detected as early as possible. Quality-of-service information is maintained on a per-session basis and hence the protocol is able to accommodate simultaneous object interactions over differing networks (e.g. if a client is talking to two services one of which is located on a mobile host while the other is on a high-bandwidth fixed network).

In addition to using quality-of-service information for tuning purposes QEX is also able to provide feedback to applications on the state of the underlying communications channels. To facilitate this we have introduced the notion of explicit bindings into the ANSAware platform. Explicit bindings are established using a bind operation which takes as parameters the source and sink interfaces to be bound and a further set of parameters which express the desired quality-of-service. Clients are returned a binding control interface as a result of the bind operation through which they can register for call-backs if the specified quality-of-service is violated. These call-backs are generated by QEX based on the information it collects for tunning purposes and allow applications to adapt to changes in the network characteristics. In this way applications can provide feedback to users on the state of the network and congestion control strategies can be adopted by applications and users in addition to the underlying protocol.

QEX has been largely developed using the network emulator which has allowed us to simulate rapid fluctuations in network quality-of-service and thus refine our algorithm for calculating retry rates. More details on QEX can be found in [Davies,94].

5. Performance

We have tested the accuracy of the network emulator over a range of different network speeds and with varying numbers of clients transmitting different packet sizes. The graphs in this section can be used to ascertain the optimum configuration file settings for a given combination of network speed and average packet size. All of the figures were taken using a network of Sun Sparx1 machines running SunOS 4.1 and interconnected using Ethernet. The emulator ran on a separate machine to the clients and servers and all the machines and the network were 'lightly loaded' at the time of testing.

To obtain the figures we ran simple client/server pairs in which the client repeatedly sent fixed size buffers to the server. The server recorded the time taken to receive a set number of these buffers and from this timing information calculated the average throughput. Standard Unix timing facilities were used throughout.

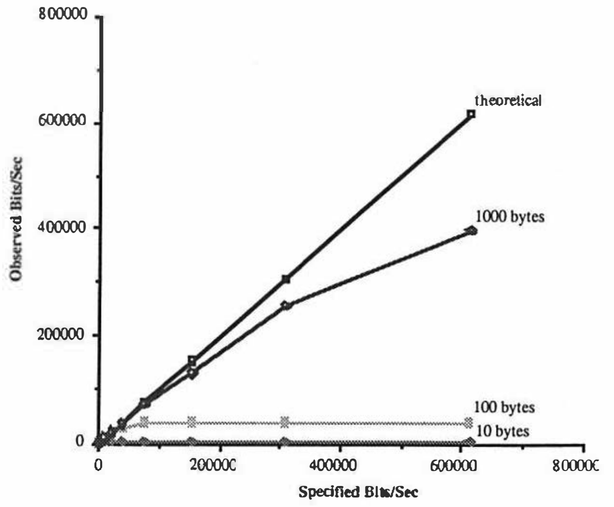


Figure 8 : Network Emulator Performance For Single Client/Server Pair in the Range 0-614400

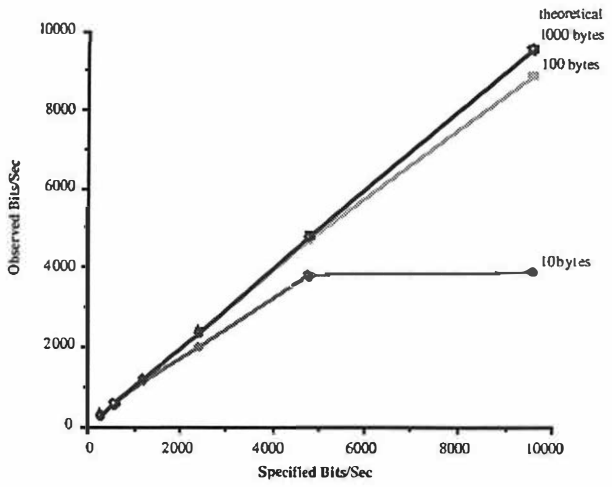


Figure 9 : Network Emulator Performance For Single Client/Server Pair in the Range 0-9600

Figures 8 and 9 shows the emulator's performance for a single client/server pair of processes. Figure 9 is based on the same timings as figure 8 but the graph shows a narrower range of network bandwidths in order to improve the level of detail which can be observed. In both graphs the x-axis is the bandwidth as specified in the configuration file and the y-axis is the observed bandwidth. The different lines denote different packet sizes (10, 100 and 1000 bytes).

The key thing to note from these graphs is that the accuracy with which the emulator models the network bandwidth is heavily dependent on the packet size. Moreover, for any given packet size there is a maximum speed at which the emulator can process and dispatch the packets. Increasing the bandwidth in the configuration file has no effect on the observed bandwidth above this cut-off point. In our tests the cut-off points were as follows: the maximum observable throughput with 10 byte packets was 3998 bytes; the maximum observable throughput with 100 byte packets was 39978 bytes and the maximum observable throughput with 1000 byte packets was 399792 bytes.

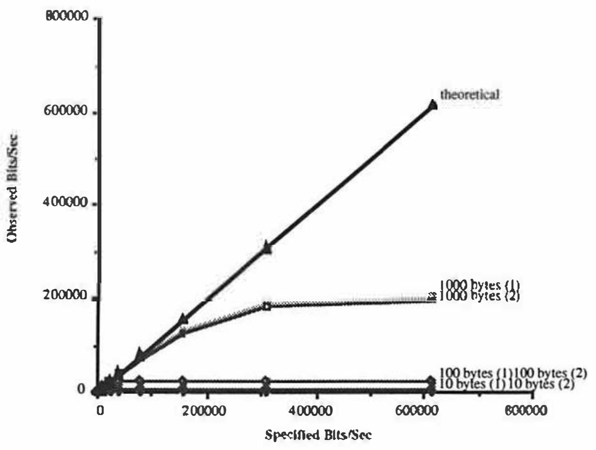


Figure 10 : Network Emulator Performance For Two Client/Server Pairs in the Range 0-614400

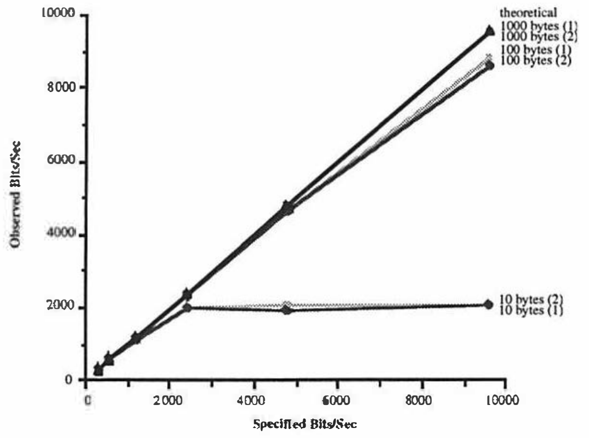


Figure 11 : Network Emulator Performance For Two Client/Server Pairs in the Range 0-9600

Figures 10 and 11 illustrate how the performance of the emulator degrades with the addition of a new client-server pair. For these figures the emulator was driven by two clients, both sending fixed size packets at their maximum rate. The graphs show the two different traces (one for each client) for the same packet sizes as above. Once again the cut-off points are evident with the maximum observable throughput with 10 byte packets being 2439 bytes; the maximum observable throughput with 100 byte packets being 19985 bytes and the maximum observable throughput with 1000 byte packets being 201000 bytes. As might be expected while the addition of new client/server pairs impacts on the performance of the emulator this impact is evenly distributed between the client/server pairs such that both see an almost identical (though less accurate) throughput.

The frequency with which the situation depicted in figures 10 and 11 occurs is clearly application dependent. In our work at Lancaster we have been focusing on the development of collaborative mobile applications for use by field engineers in the utilities industries. As a result, we have been mainly interested in emulating the type of low-speed radio networks suitable for wide-area use. In addition, the collaborative applications we have written typically have a fairly well-defined request-reply style interaction based on packet sizes of around 100 bytes and as a result we typically do not have multiple processes transmitting large numbers of messages concurrently. For this type of application the emulator has proved more than adequate and enabled us to make substantial progress in application development prior to obtaining wide-area mobile communications hardware. For more demanding applications with multiple nodes transmitting concurrently the emulator's performance can be improved by replication. In the degenerate case a separate network emulator can be used for each source node. In this case however, the emulator is only able to provide functionality equivalent to that found in Delayline since there is currently no mechanism defined for separate instances of the emulator to communicate in order to support packet collisions etc. Experimentation would be required to determine if such a distributed co-ordination protocol could be implemented while still allowing the emulators to function at level significantly better than a centralised version.

6. Concluding Remarks

This paper has described a network emulator developed at Lancaster to enable research into adaptive applications. It should be stressed that the system described provides an *emulation* of low-speed networks *not a simulation*, i.e. real applications can be compiled and executed using the emulator and these applications will experience a level of network service similar to that which they would experience if they were running over real low-speed networks. The design and implementation of the emulator has been described as has the design and implementation of a separate graphical front-end and monitoring tool for the emulator.

The performance of the emulator has been evaluated and those applications for which the emulator is best suited identified. In particular, the impact of small message sizes on the emulator's accuracy has been discussed.

The emulator and its front end have been successfully compiled and run on SUN Sparcs running SunOS, SUN Sparcs running a soft real-time version of SunOS 4.1 [Hagsand,94] and portable 486 PCs running SVR4. Sources for the emulator and the front-end are available via anonymous ftp from ftp.comp.lancs.ac.uk. In addition, the URL:

http://www.comp.lancs.ac.uk
/computing/users/nigel/emulator.html

provides more information on the network emulator and access to the source code for both the emulator and the interface described in this paper.

Acknowledgements

The work described in this paper was initiated by one of the authors while a visiting researcher at the Swedish Institute of Computer Science. The authors would like to acknowledge the contribution of Olof Hagsand who wrote the code to enable the emulator to use SICS's soft real-time SunOS and Steve Pink for discussions during the development of this emulator.

References

[APM,89] A.P.M. Ltd. "The ANSA Reference Manual Release 01.00", APM Limited, UK. March 1989.

[Duchamp,92] Duchamp, D. "Issues in Wireless Mobile Computing." Proc. Third Workshop on Workstation Operating Systems, Key Biscayne, Florida, U.S., 1992. IEEE Computer Society Press, Pages 2-10.

[Davies,94] Davies, N., G.S. Blair., K. Cheverst and A. Frdiay "Supporting Adaptive Services in a Heterogeneous Mobile Environment." Proc. 1st International Workshop on Mobile Computing Systems and Applications, Santa Cruz, U.S., December 1994.

[Hager,93] Hager, R., A. Klemets, G.Q. Maguire, M.T. Smith, and F. Reichert. "MINT - A Mobile Internet Router." Proc. IEEE VTC'93, Secaucus, NJ, U.S., 1993.

[Hagsand,94] Hagsand, O., and P. Sjödin. "Workstation Support for Real-time Multimedia Communications." Proc. USENIX Winter 1994 Technical Conference, 1994.

[Ingham,94] Ingham, D. B. and G. D. Parrington "Delayline: A Wide-Area Network Emulation Tool." Technical Report, Department of Computer Science, University of Newcastle, Newcastle upon Tyne, NE1 7RU, U.K.

[ISO,92] ISO. "Draft Recommendation X.901: Basic Reference Model of Open Distributed Processing - Part1: Overview and Guide to Use", Draft Report 1992.

[Katz,94] Katz, R.H. "Adaptation and Mobility in Wireless Information Systems." IEEE Personal Communications Vol. 1 No. 1, Pages 6-17.

[Schilit,94] Schilit, B., N. Adams and R. Want "Context-Aware Computing Applications" Proc. 1st International Workshop on Mobile Computing Systems and Applications, Santa Cruz, U.S., December 1994.

A Programming Interface for Application-Aware Adaptation in Mobile Computing

Brian D. Noble, Morgan Price, and M. Satyanarayanan
School of Computer Science
Carnegie Mellon University
{bnoble,mprice,satya}@cs.cmu.edu

Abstract

Mobile clients face wide variations in network conditions and local resource availability when accessing remote data. Coping with this uncertainty requires the ability to retrieve and present data at varying degrees of *fidelity*. In this paper we present *applicaton-aware adaptation* as a solution to this problem. The essence of our solution is a collaborative partnership between applications and the operating system. We describe the *Odyssey* API for application-aware adaptation and demonstrate its use in accessing two types of data: video and maps.

1. Introduction

Mobile clients face many challenges in accessing data from servers. Because a mobile client has to be compact and lightweight, it is typically resource-poor relative to a desktop client. Network connectivity, especially via wireless media over a large area, tends to vary considerably in bandwidth, latency, reliability and cost. Power management considerations often require certain actions to be deferred, avoided or slowed down to prolong battery life. The relative costs of accessing distributed services changes as mobile clients move. Finally, the very nature of mobility has a negative impact on robustness and security.

As a consequence of these constraints, the mechanism for mobile data access has to be *adaptive* in nature, dynamically conforming to the limitations of individual clients and their current environments. We believe that such adaptation can best be performed by a collaborative partnership between the operating system and individual applications. We refer to this strategy as *application-aware adaptation*[10].

This research was supported by the Air Force Materiel Command (AFMC) and ARPA under contract number F196828-93-C-0193. Additional support was provided by the IBM Corporation, Intel Corporation and AT&T Corporation. The views and conclusions expressed in this paper are those of the authors, and should not be interpreted as those of the funding organizations or Carnegie Mellon University.

Application-aware adaptation characterizes the design space between two extremes. At one extreme, adaptivity is entirely the responsibility of individual applications. This means that there is no focal point in the system to resolve the potentially incompatible resource demands of individual applications. It also means that there is no way to enforce limits on resource usage. At the other extreme, adaptivity is completely subsumed by the system. Although the feasibility of this approach has been demonstrated in systems such as *Coda*[5, 9], there are limits to its applicability. In particular, the end-to-end argument[8] suggests that there will be circumstances where only an application can determine the best form of adaptation. Unless the system is extended to incorporate specific knowledge about every application, there will be situations where adaptation by the system will be inadequate or even counter-productive. By striking a balance between these extremes, application-aware adaptation offers a more promising approach to mobile data access. It permits individual applications to determine how best to adapt, but allows the system to retain management of key resources and enforcement of decisions regarding their usage.

How can application-aware adaptation be effectively supported? This paper is a status report on our work toward answering this question. This work is being done in the the context of *Odyssey*, an experimental Unix platform for mobility. We have implemented a preliminary prototype and have demonstrated its use in two applications accessing data in a mobile environment. While rudimentary in many respects, our prototype does provide initial evidence of the feasibility and effectiveness of application-aware adaptation.

We begin the paper by introducing the concept of *data fidelity* and discussing the central role it plays in application-aware adaptation. Next, we discuss a number of factors influencing our design. We then describe the design of Odyssey, focusing specifically on its support for application-aware adaptation. Finally, we describe the implementation and status of our prototype.

2. Data Fidelity

Under ideal circumstances, the data presented at a mobile client should be identical to the current server copy. As resources become scarce, it may no longer be feasible to completely preserve this correspondence; some form of degradation is unavoidable. How does one characterize the extent of this degradation? We define *fidelity* as the degree to which a copy of data presented for use matches the reference copy.

Fidelity has many dimensions. One well-known, universal dimension is consistency. Other dimensions depend on the type of data in question. For example, video data has at least two additional dimensions: frame rate and image quality of individual frames. Spatial data, such as topographical maps, have dimensions of minimum feature size or resolution. For telemetry data, appropriate dimensions include sampling rate and currency.

The dimensions of fidelity are natural axes of adaptation for mobility. But the adaptation cannot be solely determined by the type of data; it also depends on the application. As we show in the next section, different applications using the same data may make different tradeoffs among dimensions of fidelity.

2.1. Video Data in Mobile Environments

Consider a movie stored on a server, and two applications accessing that video stream from a mobile client. The first application is a video playback application, *player*, and the second, *editor*, is a video scene editor. These two applications must make different fidelity tradeoffs in accessing the same video stream. No single policy can satisfy them both.

The player's primary goal is to preserve correspondence between movie time and real time. A secondary goal is to play the movie at the original frame rate, resolution, and image quality. In times of plentiful resources, the player can indeed meet both goals. However, when network bandwidth becomes scarce, the player may have to sacrifice its secondary goal in order to meet its primary goal. Thus, it may choose to switch to a black-and-white stream at full frame rate, to drop frames, or otherwise reduce the bandwidth requirements of the stream. To guard against total disconnection, the player may even hoard a very low quality version of the movie.

The editor's main goal is very different from that of the player; it must ensure that the user sees every frame of the video stream to allow precise editing. To allow this, the editor is willing to relax the correspondence between movie time and real time. Thus, when network bandwidth decreases, the editor will access the movie at a rate slower than real time to avoid dropping frames.

It is hard to see how any single operating system policy can adequately service both of these applications' needs, even though they are accessing exactly the same data. Regardless of the system's decision to change the fidelity of the stream it is retrieving, either the player or the editor – and quite possibly both – will not be satisfied. No system can be clever enough to anticipate and satisfy every application's needs. On mobile machines, where the environment is unpredictable, such unsatisfactory service will be even more evident.

3. Design Considerations

What is required to support application-aware adaptation? Generally, the system must provide a set of API extensions that allow applications to track and react to their environment, and a system architecture which effectively supports these extensions. In the sections below, we outline the desired properties of the API extensions and supporting architecture.

3.1. API Extensions

In order for applications to make decisions based on their environment, they must be able to name aspects of the environment that are important to them. This naming mechanism must be both simple and extensible. Applications should be able to specify exactly those features of the environment in which they are interested, and be notified of changes to just those features. Such specification and notification should be efficient. They must also fit into the programming style and culture of the base operating system, but cannot depend on esoteric features. Popular applications run on an increasingly diverse set of operating systems; providing common adaptation facilities enhances the portability of such applications.

As applications track changes in the environment, they must adapt their access to data. Some types of adaptation will require changes in operating system policy. There must be an efficient, flexible, and extensible mechanism to request such changes. Since the operating system is the final arbiter of resource usage, the request need not always be honored.

3.2. Supporting Architecture

What of the underlying architecture supporting these extensions? The overriding goal is *simplicity*. We are not trying to invent a new operating system, but merely extend existing ones in simple ways. We have striven to keep such extensions minimal, while making them powerful enough to explore application-aware adaptation for a wide range of data types.

It is important to note that we do not attempt to provide *resource guarantees* to applications. Such guarantees, typically encountered in real-time systems, require guarantees from lower layers of the system. But the environment of a mobile computer is too unpredictable for such guarantees. Hence, we only promise to inform applications when their environment changes, and arbitrate between applications competing for scarce and unpredictable resources.

Finally, our architecture should adhere to sound principles of software engineering. Some functionality in support of the API will be independent of the type of data, while other functionality will be type-specific. The architecture should provide isolation between different types of data as well as between the generic and type-specific portions of the system.

4. Odyssey API

This section describes our design of the Odyssey API supporting application-aware adaptation. As discussed above, there are three components to the Odyssey API. First, there is a way for applications and the system to talk about salient features of the environment. Second, there is a mechanism that enables applications to track their environment. Third, there is a mechanism through which applications request policy changes based upon their environment.

4.1. What is an Application's Environment?

We consider the salient features of an application's environment to be the *resources* available to that application. Such resources can be either *generic* or *type-specific*. Generic resources have meaning for all items stored in Odyssey. Examples of generic resources include network bandwidth between the mobile client and the server storing an item, available disk space on the mobile client, and battery power remaining on the mobile client. The generic resources in Odyssey are listed in Figure 1.

Type-specific resources have meaning only for items of a particular type. For example, consider a commercial database that indexes items in the World Wide Web. Such a service might sell a *subscription* that enables a client to make some number of queries per day. The number of queries left in a given day is a resource that is sensible only in the context of queries against that database.

Odyssey tracks and reports the *availability* of a resource, and how that availability changes. We measure the availability of an individual resource with a single scalar value. The units of a particular resource's availability are chosen appropriately for that resource. For example, network bandwidth is measured in bits per second. Available disk space is measured in kilobytes. Power remaining to a laptop is measured in minutes of operation.

Some resources are estimated with respect to a particular item in the Odyssey store. We call such items *reference items*. For example, network bandwidth between a mobile client and a server differs for different servers. Thus, we only speak of network bandwidth with respect to a particular reference item; the bandwidth in question is that between the client and the server storing that particular item. Since type-specific resources only have meaning for items of a particular type, they always have reference items.

Resource	Units	Reference Item?
Network Bandwidth	bits per second	yes
Network Latency	microseconds	yes
Disk Cache Space	kilobytes	no
CPU	SPECints available	no
Power	minutes of computation	no
Money	cents	no

This figure lists the generic resources defined for the Odyssey system. The first column lists the name of the resource. The second column gives the units in which the resource is measured. The third column specifies whether or not the resource is measured with respect to a particular item in the Odyssey store. Of particular interest is the last item, *money*. Many experimental implementations of electronic money as well as systems that use money in exchange for services exist. We believe that such services, particularly those which offer some sort of query facility, will become more common. Note that these are only the generic resources; there may be others that are type-specific.

Figure 1: Generic Resources in Odyssey

4.2. How to Track the Environment?

For an application to track the availability of resources two things must happen. First, the application must inform the system of the resources in which it is interested. Second, the system must monitor the availability of resources, and notify the application when the availability of one or more relevant resources changes in an interesting way. For efficiency, we chose to use asynchronous notification rather than polling in Odyssey.

Naturally, not all applications will be interested in the same set of resources. To tell the system what resources an application is interested in, the Odyssey API provides a call, `ody_request`. For example, an application making an `ody_request` might ask, "Please invoke procedure `bar` if the network bandwidth between here and the server storing `/ody/foo.c` exceeds ten Mb/s or falls below one Mb/s." The C declarations for `ody_request` and associated data structures appear in Figures 2 – 4.

Requests name the *resource* of interest, the *bounds of tolerance* on that resource's availability, the *reference item*, and an *upcall procedure*. In our example above, the resource of interest is network bandwidth. The upper tolerance bound is ten Mb/s, and the lower bound is one Mb/s. The reference item is `/ody/foo.c`, and the upcall procedure is the procedure `bar`.

The resource is named in the `ody_req_des_t` structure, as are the tolerance bounds and the address of the upcall procedure, which is a handler function much like a signal handler. The resource is named by an integer identifier. Generic resource identifiers are known throughout the system; type-specific identifiers are known only to portions of the system that implement that type, but are limited to a specific range. If the resource is not within the specified tolerance bounds, the call fails and returns the current value in `res`. Otherwise, the request is registered with the system. If the resource later strays outside of those bounds, the system invokes the handler through an upcall.

If a request is granted by the system, the system returns a request identifier. That request identifier is also passed to the request handler when the application is notified by the system. If the application no longer wishes to be notified for that request, it can invoke `ody_cancel` on it.

4.3. How to Request Policy Change?

As applications are notified of resource changes, they will need to adapt their access patterns. Some of this adaptation will require changes in policy within the operating system. Since policies are type-specific, these requests for changes in policy must also by type-specific. We call such a request a *type-specific operation*, or `ody_tsop`. An example of a type specific operation would be, "Please switch from the full color version of this stream to the black-and-white version."

Just as there is no way to predict the needs of all applications , there is also no way to predict all possible requests for policy changes. So, instead of trying to enumerate them for each type *a priori*, we instead provide a general mechanism to allow for experimen-

```
/* Pathname resource request */
int ody_request (path, req, res);
char           *path;          /* pathname of reference item */
ody_req_des_t  *req;           /* A request descriptor */
long           *res;           /* The request id returned, or current value */

/* Cancel a request */
int ody_cancel (reqid);
long reqid;              /* The request to cancel */
```

This figure shows the C declarations for the pathname-based version of ody_request, as well as ody_cancel. The descriptor-based version is identical except that a file descriptor is used instead of path. Note that ody_request is similar to the UNIX sigvec system call. ody_request allows an application to place a notification request req; ody_cancel cancels an outstanding request. Declarations for relevant data structures can be found in Figure 3; the signature for the callback function to be invoked on notification of an outstanding ody_request is shown in Figure 4.

Figure 2: C Declaration for ody_request and ody_cancel

```
/* A version stamp*/
typedef struct {
    long        gs;         /* Version of generic resource interface*/
    ody_codex_t codex;      /* The type of the reference item*/
    long        cs;         /* Version of type-specific resource interface*/
} ody_vers_t;

/* A resource request descriptor */
typedef struct {
    long           resource;     /* Resource identifier */
    ody_vers_t     version;      /* Version stamp */
    long           low, high;    /* low, high values of window */
    ODY_REQ_FN_T   fn_ptr;       /* function to call if window is left */
} ody_req_des_t;
```

These are the principal data structures used in the ody_request call. ody_vers_t is used to ensure that the application and system are using the same set of resource identifiers, and that the application and the system agree on the type of the reference item. The type ody_codex_t is an enumeration of known types in the system, called *codices*. The req_des_t type holds the fields of a request: the resource, version information, the window of tolerance, and the upcall procedure. The signature for upcall procedures is shown in Figure 4.

Figure 3: Data Structures for ody_request

```
/* A resource request handler */
typedef void (*ODY_REQ_FN_T)(long, long, long);
/* the three arguments are: */
/*      the request id to which this notification is responding */
/*      the resource identifier */
/*      the current value of the resource */
```

This figure shows the type signature of a request handler. A request handler takes three arguments: the request identifier, as returned by ody_request, to which this notification is responding, a resource identifier denoting the resource that has changed, and the new availability of that resource.

Figure 4: Notification Handler Declaration

```
/* Pathname-based type specific operation */
long ody_tsop (path, vers, op, argsz, arg, retsz, ret);
char        *path;      /* pathname of reference item */
ody_vers_t   vers;      /* version of this codex' interface */
long         op;        /* which operation to perform */
size_t       argsz;     /* size of argument buffer */
void        *arg;       /* arguments for operation */
size_t       retsz;     /* size of return buffer */
void        *ret;       /* return buffer */
```

This figure shows the C declaration for ody_tsop, the pathname-based invocation of a type-specific operation. The descriptor-based version is identical except that a file descriptor is used instead of path. The arguments name the reference item, version information, the operation to be performed, and buffers for the arguments and results. The definition of ody_vers_t can be found in Figure 3. The sizes of the argument and result buffers must be passed, so that layers that do not know the details of the particular type can pass arguments correctly. Note that this is similar in flavor to the UNIX ioctl system call.

Figure 5: C Declaration for ody_tsop

tation and extension. The C declaration for ody_tsop appears in Figure 5.

To invoke ody_tsop, an application must specify a reference item. It must also specify the operation to perform, the arguments to the operation, and a buffer for the return value. The type of the reference item determines the type of the ody_tsop, and the reference item is passed through to the body of code that implements the ody_tsop. The reference item can be specified by file descriptor or pathname. The operation is denoted by an integer identifier, and need only be unique within a single type, thus preserving independence between different types. The sizes of these buffers are specific to the operation; hence, they are also type-specific. To preserve isolation between generic and type-specific portions of Odyssey, the sizes of these buffers must be specified.

The type-specific operation mechanism is designed to allow applications to make policy requests. However, once it is present, ody_tsop can be leveraged to provide a set of access methods richer than the simple file system interface provided by common operating systems. For example, items of type "video" might support the type-specific operation video_read_frame, which reads a single variable sized frame, in addition to the simpler read system call. Such extension allows us to use data of different types in ways that are natural to the data, rather than forcing the data to fit the more restrictive file system model.

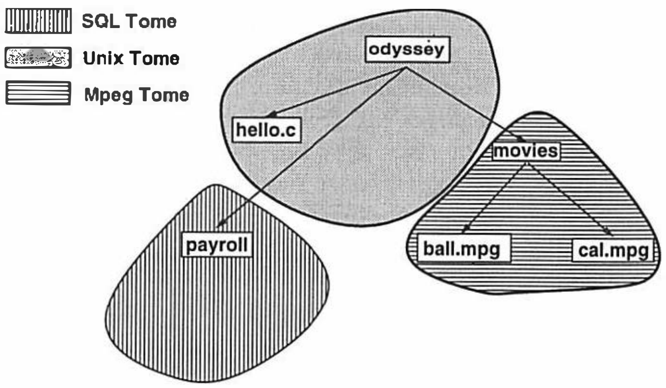
5. Odyssey Structure

To support the Odyssey API, our design provides three extensions to UNIX. First, we have added a notion of *type* to the standard UNIX file system. Second, we have added a generic cache manager, the *viceroy*, to provide type-independent support for the Odyssey API. Third, we have provided a set of *wardens*, which are part of the Odyssey cache manager, each providing support for an individual type in the Odyssey store. The next three sections explore each of these in turn.

5.1. Adding Types to the Operating System

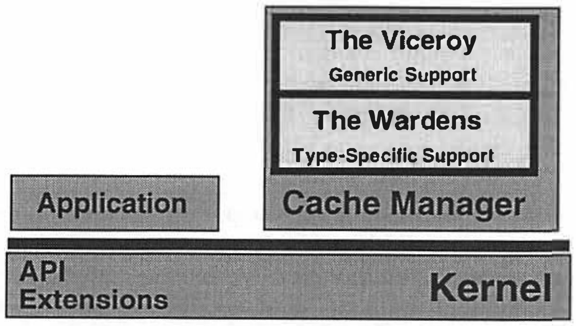
Odyssey provides a single, global namespace to its clients. A simple example of such a namespace is shown in Figure 6. This namespace is broken into subspaces called *tomes*, or *typed volumes*. Tomes are similar to *volumes* in AFS and Coda[11, 4, 9]. A tome carries with it a notion of type; all items in a tome are of the same type. A tome's type determines type-specific resources, operations, and dimensions of fidelity for items in that tome. All tomes which have the same type are logically grouped together into a *codex*.

We envision a small number of types in Odyssey. The implementation effort to add a type is nontrivial, and will likely be undertaken by experienced system builders. A new type will be justified when applications using data of that type exhibit access patterns fundamentally different from any other existing ones. In the video example in Section 2.1, the player and editor have roughly the same access patterns, but prefer



This figure illustrates a sample Odyssey namespace. In this example, there are three *tomes*, each of a different type. The first tome, rooted at `odyssey`, contains the single UNIX file `hello.c`. The second, rooted at `payroll`, is a database. Note that no nodes appear inside of `payroll`; it is named associatively rather than hierarchically. The third tome, rooted at `movies`, contains two MPEG movies, `ball.mpg` and `cal.mpg`.

Figure 6: Odyssey Tomes



This figure illustrates the architecture of an Odyssey client. Odyssey applications make use of the Odyssey API extensions along with the operating system's API. Operations on Odyssey objects are redirected by the kernel to the cache manager, which is at user level for ease of implementation. The cache manager is split into two logical pieces: the viceroy, providing generic support, and a set of wardens, each supporting a single type.

Figure 7: Odyssey Client Architecture

to make different tradeoffs. In contrast, video data, which is inherently linear, will be accessed differently from topographical maps, which are inherently spatial.

5.2. Providing Generic Support

There are many client-side tasks that are independent of data type. This generic functionality is implemented by the *viceroy*. The viceroy can be thought of as the generic cache manager, which depends on type-specific cache managers to do its job. The logical role of the *viceroy* is illustrated in Figure 7.

The viceroy's most important task is to act as the single point of resource control in the system; all other pieces of the Odyssey client are subordinate to it. The viceroy also handles requests for generic resources, and notifies applications when those resources leave requested bounds. Finally the viceroy responds to requests on individual Odyssey objects, and forwards them to the appropriate warden.

5.3. Providing Type-Specific Support

We call Odyssey's type-specific cache managers *wardens*. There is one warden in the Odyssey cache manager for each type in the Odyssey store. The wardens' logical role on the client is illustrated in Figure 7.

The wardens are responsible for implementing the access methods on objects of their type – both the standard UNIX operations as well as the type-specific ones. The wardens also implement a number of different fidelity mechanisms, and allow applications to choose between them. In addition, they provide reasonable default policies for naive applications. Default policies are also important in providing backward compatibility with legacy applications.

6. Implementation Status

We have built a preliminary prototype of the Odyssey client along with applications, wardens and servers for two data types. The goals of the prototype were twofold. First, we wanted to code applications that might benefit from application-aware adaptation to the Odyssey API to test the efficacy of the interface. Second, we wished to explore the practical implications of the division between viceroy and warden.

The two data types we have explored are QuickTime[1] and GRASS[7]. QuickTime is a multimedia encoding standard proposed by Apple Computer. GRASS is a public domain geographical information system. Along with some basic applications using these data types, we provide a simple control program to a user of the prototype. The control program is used to simulate various network bandwidths on the connection between the cache manager and various servers. The applications then change the fidelity of the data they access to match the simulated bandwidth. While each application works well in isolation, we have not yet explored resource control mechanisms to arbitrate between them.

The QuickTime application we have explored is a movie player. The player can open a QuickTime movie on a server via the Odyssey cache manager and begin playing it. The server stores the movie at several different levels of fidelity, and bundles them into a logical movie. The player, by using `ody_request` and responding to notifications, asks the cache manager to fetch the highest fidelity stream that can be played in real time given the available bandwidth.

The GRASS prototype supports applications via a modification to the GIS library. These applications display, query, and combine geographical data. The main type of data is raster data: a two-dimensional array of values set into a coordinate space. The client caches files from the server in the local filesystem; the raster data is fetched at various resolutions, depending on available network bandwidth. The GRASS applications then access those cached files.

We have made many simplifications for ease of rapid prototyping. The current prototype is completely user-level, trading realistic resource management policies and performance for simple implementation. It makes no attempt to measure resources, and depends on the control program instead. The UNIX file system call interface is not currently implemented; the application uses the Odyssey API exclusively in communicating with the viceroy, and uses the local file system when necessary for a cache. The prototype consists of a library linked into Odyssey applications, a prototype cache manager and wardens, and the applications and servers. Each of these is discussed below.

6.1. The Odyssey Library

The API extensions are provided by a library linked with the prototype application. All of the calls described in Section 4 are provided, but the prototype does not include the standard file system interface. The library communicates with the cache manager via RPC. The library responds to all notifications by the prototype cache manager, and forwards them to the proper upcall handler registered by the application; the UNIX `signal` interface is used to simulate upcalls.

6.2. The Prototype Cache Manager

The prototype cache manger consists of a simple viceroy, along with the QuickTime and GRASS wardens. It performs minimal resource management, and makes no attempt to authenticate users or arbitrate between conflicting applications. Rather than attempt to estimate resources, it depends on the external control program to advise it. It implements `ody_request` and `ody_cancel`, and forwards `ody_tsop` operations to the wardens based on the reference item's type. It notifies applications by sending them a signal, and passing information about the notification through the filesystem. In the sections below, we describe both the QuickTime and GRASS wardens.

6.2.1. The QuickTime Warden

The QuickTime warden exports the interface we envision for the final system. It has no type-specific resources, but has four type-specific operations. Those operations are shown in Figure 8. `QT_SwitchTracks`, is a request for policy change, while the other three perform data access. Each of these operations is explained below.

`QT_OpenMovie` takes a string which represents a movie name and attempts to open it at every available fidelity level on the server. Each version is opened as a *track* of the base movie, and they are logically bundled as a single movie and returned. Along with a handle for the movie, akin to a file descriptor, `QT_OpenMovie` returns information about each track – specifically, the average bytes per second required to transmit each track across the network and the encoding method of each track. Upon opening, the best possible track is made the *active track*, and will remain active until the application requests otherwise. `QT_CloseMovie` frees up any resources associated with an open movie.

`QT_GetFrame` takes a movie handle, returned by `QT_OpenMovie`, and a time offset into the movie, and returns the first frame of the active track to be displayed

`QT_OpenMovie(m)`	Open movie m and return track information.
`QT_CloseMovie(m)`	Close movie m and free resources.
`QT_GetFrame(t)`	Returns the first frame to display after time t.
`QT_SwitchTracks(m, i)`	Ask to make track i of movie m the active track

Figure 8: Operations Supported by the QuickTime Warden

after the offset. The frame is copied into the `ody_tsop` return buffer for use by the player. `GetFrame` also returns the index of the currently active track, so the application can properly decode the frame.

`QT_SwitchTracks` takes a movie handle, and a track identifier within that movie handle, and makes that track the new active track. Readahead is terminated for the old active track, and started for the new track. After the pre-read portion of the old track is exhausted, `QT_GetFrame` will return a frame from this new track. The new track will be used until another `QT_SwitchTracks` request is made.

6.2.2. The GRASS Warden

The GRASS warden provides two operations: `Grass Fetch`, which fetches a logical file from a server if not already cached, and `GrassSetQuality`, which determines which fidelity level future fetches will use. The final version of the system won't need `GrassFetch`: it'll have `open` redirected to it instead.

GRASS stores logical files in groups of related physical files. To avoid inconsistencies such as a raster header file showing the full size and a raster data file with lower resolution data, the prototype warden fetches files as a group. The GRASS warden currently makes no effort at cache replacement. Future refinements will address this as well.

6.3. The QuickTime Server and Player

The obvious fidelity dimension to exploit in video is the quality of individual frames; by reducing frame quality, we can also reduce bandwidth requirements. The QuickTime server currently stores movies at three different fidelity levels: full color uncompressed, full color with lossy JPEG compression, and black and white. Individual tracks can be opened, pre-read and closed. The server itself does not manage the different fidelity levels of the same logical movie as a unit; that is handled by the QuickTime warden.

The QuickTime player was modeled after a previously built standalone version that used the UNIX file system interface. It was redesigned to use the `ody_tsop` interface exported by the warden, rather than the standard UNIX file system interface. The new player opens a movie, finds the stream with the highest possible quality, and begins playing it. It also places a request to be notified if the bandwidth drops too low to support this track. If so, it switches to the new best possible stream. If, at some later time, bandwidth improves enough to allow playing a better track, the player will request a change.

Although the prototype explicitly trades performance for ease of implementation, the player has adequate performance in playing back movies, even at the highest quality. Of particular interest is the fact that the player was both simplified and functionally improved by the switch from the UNIX file system interface to that provided by Odyssey.

6.4. The GRASS Server and Applications

The server stores raster objects at three levels of fidelity, losing a factor of two in resolution for each degradation. Because the rasters are two dimensional, each degradation provides a savings of a factor of four in data size.

Applications wishing to open raster objects share a single routine in the GIS library. That routine first determines the estimated bandwidth available to the viceroy through the request interface with an empty bounds window, effectively polling the viceroy. Since no value could satisfy that bounds window, the bandwidth estimation is returned by the request call. The application then uses the `GrassSetQuality` operation to ask for a particular fidelity of raster. That fidelity is then cached on local disk for future use by GRASS applications.

7. Conclusion

Though rudimentary in many respects, our preliminary prototype has allowed us to gain initial validation of our ideas at low implementation cost. The results so far are encouraging. We have taken the source code of applications for two data types and have been able to restructure them into the Odyssey framework with modest effort.

We are now working toward a more complete and efficient prototype, motivated by two goals. First, we would like the prototype to support a broader collection of data types and associated applications. This will stress the designs of the Odyssey API and architecture, expose shortcomings, if any, and lead to refinements in both. It will also deepen our understanding of application-aware adaptation. Second, we would like the prototype to be better integrated with an operating system. An in-kernel implementation will allow more serious resource management, provide better performance and functionality, and enable more rigorous evaluation of our design.

As was discussed early in this paper, the constraints of mobile computing lead inevitably to the recognition that adaptivity is essential in any system that provides mobile data access. But although the general importance of adaptivity has been recognized by many researchers[2, 3, 6, 12, 13], we are not aware of specific system designs, much less implementations, that support application aware adaptation. The work reported here thus represents a journey into uncharted waters.

References

[1] APPLE COMPUTER, INC. *Inside Macintosh: QuickTime.* Addison-Wesley Publishing Company, 1993.

[2] DUCHAMP, D. Issues in Wireless Mobile Computing. In *Proceedings of the Third Workshop on Workstation Operating Systems* (Key Biscayne, FL, April 1992).

[3] FORMAN, G. H., AND ZAHORJAN, J. The Challenges of Mobile Computing. *IEEE Computer 27*, 4 (April 1994).

[4] HOWARD, J. H., KAZAR, M. L., MENEES, S. G., NICHOLS, D. A., SATYANARAYANAN, M., SIDEBOTHAM, R. N., AND WEST, M. J. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems 6*, 1 (February 1988).

[5] KISTLER, J. J., AND SATYANARAYANAN, M. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems 10*, 1 (February 1992).

[6] KULKARNI, D. C., BANERJI, A., CASEY, M. R., AND COHN, D. L. Information Access in Mobile Computing Environments. Tech. Rep. TR-93-11, University of Notre Dame, Notre Dame, 1993.

[7] MADRY, S. Geographical Resources Analysis Support System (GRASS), an Integrated Public Domain GIS and Image Processing System. In *GIS/LIS 1989 Proceedings* (Orlando, FL, November 1989).

[8] SALTZER, J., REED, D., AND CLARK, D. End-To-End Arguments in System Design. *ACM Transactions on Computer Systems 2*, 4 (November 1984).

[9] SATYANARAYANAN, M., KISTLER, J. J., KUMAR, P., OKASAKI, M. E., SIEGEL, E. H., AND STEERE, D. C. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers 39*, 4 (April 1990).

[10] SATYANARAYANAN, M., NOBLE, B., KUMAR, P., AND PRICE, M. Application-Aware Adaptation for Mobile Computing. *Operating Systems Review 29*, 1 (January 1995). Also available as Tech. Rep. CMU-CS-94-183, Carnegie Mellon University, School of Computer Science.

[11] SIDEBOTHAM, R. Volumes: The Andrew File System Data Structuring Primitive. In *European Unix User Group Conference Proceedings* (August 1986). Also available as Tech. Rep. CMU-ITC-053, Carnegie Mellon University, Information Technology Center.

[12] THEIMER, M., DEMERS, A., AND WELCH, B. Operating System Issues for PDAs. In *Proceedings of the Fourth Workshop on Workstation Operating Systems* (October 1993), IEEE.

[13] WEISER, M. Some Computer Science Issues in Ubiquitous Computing. *Communictions of the ACM 36*, 7 (July 1993), 75–84.

System Isolation and Network Fast-Fail Capability in Solaris

Gabriel Montenegro (gabriel.montenegro@eng.sun.com)

Steve Drach (steve.drach@eng.sun.com)

Sun Microsystems, Inc.

Abstract

UNIX hosts configured for network operation typically hang, or freeze, when temporarily disconnected from the network. This failure is unacceptable to the user of a mobile host who purposely disconnects the host from the network in order to move it to another location not serviced by a network connection. This paper describes an approach that automatically enables a system to continue to function, in a diminished capacity, when disconnected from the network.

1.0 Introduction - Why is there a problem?

Modern computing environments typically consist of a core cluster of servers and many desktop workstations. Together they implement a client-server model of decentralized distributed processing that depends on a robust functional network for successful operation. This type of distributed processing avoids reliance on a particular host and, instead, places the emphasis on the network. In other words, "the network is the computer."

The problem is, if a network operation fails, the system is often rendered unusable. It is assumed that this is a temporary condition, and the best recovery procedure is to periodically retry the failed network operation until conditions improve. Some clients, such as NIS, ignore network failures and retry forever, effectively preventing their clients from continuing to operate, even in a diminished capacity. In most cases, the system appears to hang or freeze during the period of disconnection.

With the advent of nomadic computing devices, systems operating with intermittent network connectivity are increasingly more common. In this case, system isolation brought on by network disconnection may very well be purposeful and expected to continue for the duration of the user's session. Any network operations are guaranteed to fail. It is not, however, desirable for the system to stop functioning.

An ancillary problem is that several applications built specifically for the mobile or nomadic user have appeared. These typically have a disconnection state, in which they either log operations to be replayed upon reconnection, or make use of local caches. Each application determines system connectivity using a private method that typically depends on time-outs. This presents the user with different non-homogeneous models of system behavior during a network outage.

It has been pointed out that the main problem with distributed systems is that the applications are implemented with the assumption that all the processes involved reside in the same system. As soon as latencies, errors and network outages occur, the paradigm breaks down [WALDO 94], and errors such as system hangs appear.

Obviously, one approach to solving the problem is to modify existing services and applications so that they have more information regarding network status and so that they are more aware of the characteristics of the underlying communications medium. This approach, however, is very expensive and time consuming.

We chose, instead, to provide a fast-fail capability so that network operations that are guaranteed to fail, do so quickly and that the clients who are notified of the failures give up quickly. Instead of waiting for network operations to resume (and hanging during the outage), it makes more sense for the system to continue operating, using whatever local resources (e.g. cached information) may be available.

In addition to allowing currently existing applications an opportunity to continue operations during a network outage, we have also provided mechanisms for developing more knowledgeable applications and kernel modules that modify their behavior during a network disconnection.

The fast-fail capability can also alleviate the ancillary user interface problem described earlier, by providing these applications with immediate and unequivocal indication of network disconnection. They no longer need to implement private algorithms to determine the state of network connections.

2.0 What is System Isolation?

When thinking about system isolation, several related concepts and networking conditions come to mind:

- stand-alone systems
- time-outs
- temporary network outages
- Mobile IP
- network partitioning
- disabled interfaces
- weakly connected systems

We now examine each of these in some detail.

The problem we are trying to solve only concerns hosts that must adapt to intermittent network connectivity. This means that a system that is always disconnected from the network is not part of the problem domain. It is a stand-alone system that does not run the risk of freezing by retrying network operations.

System isolation is a guarantee that no network access is possible, and will not be possible for much longer than usual network time-out periods, typically ranging from under a second to several seconds. Accordingly, system isolation occurs when a time-out is on the order of minutes or hours.

Admittedly, the distinction between system isolation and a temporary network outage is sometimes hazy. The determination on whether a system is isolated or not can also depend on the particular network technology. For example, wireless interfaces can experience loss of signal in between cells, because of fading, etc. These are momentary and do not represent system isolation. Mobile IP [PERKINS 95] deals with this scenario to some extent by suggesting procedures for hand-off and for reducing the adverse effects of this unreliable link.

On the other hand, mobile systems may choose to operate in isolation for extended periods of time (e.g. during travel in a plane flight). This is not handled by Mobile IP. After all, this technology assumes the existence of a network, albeit a mobile one. System isolation assumes there is no network, mobile or static, so all procedures meant to circumvent the obstacles of a lossy, noisy medium are in vain: there simply is no medium. The system cannot use any form of IP.

System isolation is also different from network partitioning. Here, a collection of hosts may be able to reach some but not all destinations in the network. Since the network is still available (albeit in a diminished capacity), this condition is best handled by level 2 routers repairing the partition [PERLMAN 92]. A host in the partitioned region may receive ICMP error messages [POSTEL 81] indicating that the destination or the net is unreachable. During this time, it makes sense for this host to continue retrying the network access. Notice that this may also be the case if some of the interfaces of a multi-homed host are disabled (perhaps as a result of `ifconfig down`).

Finally, mobile systems often disconnect from the network and reconnect at another point using a vastly different medium (e.g. moving from a high-speed Ethernet LAN to a WAN using PPP). Furthermore, in the wireless case the network characteristics fluctuate dramatically even without changing media. These scenarios imply that a system may become weakly connected (i.e., service may diminish to the point that it is no longer usable, although strictly speaking the network is still available). Even though this is not described by system isolation, our prototype includes a condition notification facility that could accommodate this scenario (see Section 6).

3.0 Our Solution

The IP fast-fail capability prevents the system from freezing or retrying network operations when it is isolated. Network client processes become nonpersistent: instead of waiting for a reply from their server, they receive a notification that they are isolated from the network, and they immediately fail the operation. There are no futile retries.

Alternatively, clients can react to system isolation by making use of local caches or recording network operations to be replayed upon reconnection.

Our prototype implements this capability within the following constraints:

1. Accessing the loopback interface must not trigger the fast-fail mechanism. Only outgoing network accesses trigger it.
2. Existing, unmodified, applications should benefit from the fast-fail mechanism.

Our prototype works very well with existing applications (see section 7). When a system becomes isolated, there is no need to change its configuration to prevent it from issuing network requests. For example, the path environment variable may still point at network resources. These path queries fail immediately by virtue of IP fast-fail detection incorporated into NFS. The automounter, in particular, may try several servers in order to mount a certain file system over the network. Usual behavior implies a hang of several minutes until, one-by-one, all the servers time out. With fast-fail, there is no delay.

We have tested applications built with system isolation in mind. Sun's ROAM nomadic mail tool and a prototype disconnected cache file system now have an unequivocal and immediate notification that the system is isolated. They respond by entering their own disconnected operation mode.

We have also provided a framework for applications to subscribe to notifications of network conditions. This is for isolation aware applications that wish to make better use of this information.

4.0 Enabling and Disabling System Isolation

The system becomes isolated either:

1. explicitly, by direct user input requesting system isolation, or
2. implicitly, when all (non-loopback) interfaces are down or unusable.

Our prototype includes both a command line and a graphical user interface tool that allows the user to explicitly change the state of system isolation, and to inquire about the isolation status.

Implicitly enabling system isolation provides a capability for automatic reconfiguration if a smart interface detects network disconnection and configures itself down. If all interfaces configure themselves down, then the system becomes isolated. Notice that new semantics take effect only when ALL non-loopback interfaces are configured down. This allows the user the ability to selectively mark some interfaces down without causing system isolation.

For the SPARCstation Voyager, we have implemented a connection monitoring daemon that takes advantage of a special hardware feature to determine whether or not the Ethernet cable is plugged in. When the cable is removed, the daemon issues an `ifconfig down` on the interface. Usually, this is the only (non-loopback) interface the system uses. Consequently, the system automatically enters the isolated state. At this point, the system become perfectly usable in a stand-alone mode.

When the ethernet cable is plugged back in, the daemon senses it, and issues the equivalent of an `ifconfig up` on this interface. Since the number of *up* interfaces is no longer 0, the system is not isolated, and normal behavior resumes. Notice that the same is true if the user issues an `ifconfig up` on any interface (e.g., a PPP interface).

On systems without this special hardware support, a similar facility can be built into the Ethernet driver itself. When it detects no carrier on the physical interface, it can cause the interface to be marked down, which in turn causes the system to enter the isolated state. When the driver later senses a carrier, the interface can be brought up, resuming normal network operations.

5.0 Reacting to System Isolation

Once the system is isolated, IP's output routines generate an ICMP error report whenever an IP packet is sent to any interface except the loopback interface. The error message is returned back up the appropriate UDP, TCP, or ICMP protocol stack to the application.

5.1 Using ICMP to Propagate the Errors

We chose to use ICMP instead of a different mechanism because typical TCP/IP code already handles a similar case upon reception of an ICMP port unreachable message, and therefore it could be implemented without significant changes to existing network code. The only remaining question was *which ICMP error should we use to flag this condition*?

The original ICMP specification defines three other codes for destination unreachable errors:

- fragmentation needed
- host unreachable
- net unreachable

Of these, the last two appear related to fast-fail during isolated operation. However they are quite commonly used, and we did not want to overload their meaning. In fact, further investigation revealed we could not use them. RFC-1122 ("Requirements for Internet hosts - communication layers") specifies why [BRADEN 89a]:

> "A Destination Unreachable message that is received with code 0 (Net), 1 (Host), or 5 (Bad Source Route) may result from a routing transient and MUST therefore be interpreted as only a hint, not proof, that the specified destination is unreachable [IP:11]. For example, it MUST NOT be used as proof of a dead gateway (see Section 3.3.1)."

In short, the host and net unreachable codes are hints and are usually treated as *soft errors*. According to RFC-1122, these are simply recorded for eventual return if the connection times out. *Hard errors* abort the connection.

Even though [BRADEN 89a] defines additional codes:

6:	destination network unknown
7:	destination host unknown
8:	source host isolated
9:	communication with destination network administratively prohibited
10:	communication with destination host administratively prohibited
11:	network unreachable for type of service
12:	host unreachable for type of service

it is not documented whether each of these should be treated as hard or soft [BRADEN 89b]. It seems like the code we could adopt for isolated operation is 8. This code was created for routers (actually, IMP's) for return to hosts [POSTEL 94].

As originally envisioned, reception of a source host isolated error from a router or IMP should be a hint. If one router informs the system that it is isolated, it does not mean that other paths (traversing other routers) are not available. However, if the source host isolated packet originates from the local system, as can be ascertained by checking the source address field, then we know the outgoing IP packet never made it into the network. This is no longer a hint, but a hard fact that the system does not have access to ANY network. In this case, we chose to treat this indication as a hard error.

There are two reasons why this use of the source host isolated error code does not introduce confusion.

1. Currently, this code is used very rarely, if at all. Not only is it considered obsolete [STEVENS 94], but current internet routers are required not to generate it [ALMQUIST 94].
2. Even if this code were received by the networking modules, it would not be interpreted as an indication of system isolation unless the source address of the ICMP packet corresponds to the local system. This limits the sender of the packet to the local system's fast-fail code.

In effect, the source address of the source host isolated packet determines the semantics to be adopted. For example, if the source is not local, 4.4BSD simply treats this message as equivalent to host unreachable. Thus, the user process displays the error message "No route to host". On the other hand, if the source is local, corresponding to the system isolation interpretation, the user sees the error message "Network is down".

5.2 Modifications to Network Modules

If there is an outgoing packet and the system is isolated, the IP module starts the chain of events by creating and returning upstream an ICMP unreachable error with code *source host isolated*.

The ICMP module was augmented to pass ICMP error notifications to the correct UDP stream. UDP handles the notification by producing a T_UDERROR_IND TLI message for consumption by its clients. Both UDP and TCP translate the ICMP error notification to the UNIX error ENETDOWN (127). Depending upon the state of a user selectable option, TCP may:

- record the ENETDOWN error to be returned when TCP finally times out, or
- kill the connection.

To do so, it uses the same function it calls to terminate timed out connections, but with the error ENETDOWN.

The sockets kernel module handles the T_UDERROR_IND message by sending an M_ERROR message up to the stream head. However, it only does so for connected sockets (i.e. those for which the source and destination information allow some sanity checks).

The next access to the socket returns with the appropriate error code, and the stream becomes unusable from this point on. The transport independent kernel module handles the T_UDERROR_IND by simply propagating it upstream.

Additionally, we made minor changes to `in.rdisc` (router discovery), `ping` and `snoop` for correct decoding of the source host isolated code. Router discovery was also made isolation aware by allowing it to bypass time-out loops if a source host isolated error is received.

6.0 A Proactive Approach

ICMP error messages certainly offer the advantage that the system already has the facilities to handle them. However, they are limiting in two important ways:

1. the type of information that can be carried
2. they are only sent in response to previous traffic.

In order to alleviate this, we have defined a set of ioctl commands for IP that allow isolation aware applications to query, set or reset the isolation status of the system. For example, NIS and DNS use the query mechanism to check the system's status before initiating network operations.

Applications may wish to be notified whenever the isolation status changes. Accordingly, a subscription mechanism was introduced for applications to receive immediate notification whenever IP's connectivity changes. We use this facility to implement a graphical application that displays real-time isolation status.

Another ioctl command with a similar purpose allows kernel modules to register a callback function with IP. When IP senses a change in isolation status, it invokes the registered function. A user-selectable option employing this facility allows the TCP module to abort all connections in the process of transmitting data.

These subscription and registration interfaces provide a framework for lower layers to notify upper layer protocols and applications of varying network conditions. For example, a cellular handoff notification could be used by TCP to "kick" its fast retransmit code [CACERES 94]. A good candidate for notification is the amount of bandwidth available. Each layer could use this information to decide if it is worth offering service. For example, if the bandwidth available is 2400 bps, TCP and UDP might still offer service, but NFS could just return an error. In effect, this allows the specification of minimum conditions for each type of network connection.

By generalizing these mechanisms we plan to implement a condition notification facility that separates *information* (e.g. "The bandwidth is now 4800 bps"), from *policy* (e.g. "What am I going to do about it?"). This gives each layer the freedom to make its own decisions. Of course, some layers depend on others. If in the above example UDP says: "This is too slow for me, I will send an error message upstream", NFS does not have much choice but to propagate an error upstream.

This enhanced condition notification facility will not only implement the current semantics for notification of system isolation and connectivity, but it will also accommodate additional conditions such as *moved* (to report mobile handoffs), *bandwidth* (for periodic updates of actual network bandwidth), *cost* (for connectivity pricing information), etc.

7.0 Application Response to Fast-Fail

One of the main motivations in our approach has been to allow unmodified applications to fast-fail. There may be two different responses depending on whether the application establishes its network connection before or after the system becomes isolated. We have tested both cases when applicable.

7.1 System Isolated before Issuing the Command

Usually, the application displays one of these two error messages:

1. Network is down
2. Unknown host

Message 1 above occurs when the application itself attempts the network operation. Usually, however, the application must first resolve a name. Depending on the system's configuration, DNS or NIS may contact a server on the application's behalf. They receive the error notification, and fast-fail in their efforts to resolve a name. The application itself also fast-fails, but this time it displays message 2 above. The results of our testing were:

- `rsh, rlogin, rcp, finger`

  Immediate exit after displaying one of the error messages above.
- `ftp, telnet`

  Immediate return to application prompt (`ftp>` or `telnet>`) after displaying one of the error messages above.

- `spray`

  Exits with an immediate failure notification to the user:

  "spray: cannot clnt_create <rhost>: netpath: RPC: Rpcbind failure - RPC: Unable to send"

  If resolving a name, the error message is:

  "spray: cannot clnt_create <rhost>: netpath: RPC: Unknown host"

- `ping`

  If resolving a name, immediate exit with message 2. Otherwise, it loops retrying to reach the target host, and displaying the ICMP host isolated error received from IP:

  "ICMP Source Host Isolated from gateway <localhost>..."

- `rusers`

  Immediate exit and message 2, if resolving a name. Otherwise, the following message:

  "RPC: Rpcbind failure"

- WWW browsers `Mosaic` and `netscape`

  Immediate notification formatted in html:

  "<H1>Fatal Error 500</H1>"

  The error may be caused by the application itself:

  "<B>Reason:</B> System call `connect' failed: Network is down."

  Or by name resolution via DNS or NIS:

  "<B>Reason:</B> Can't locate remote host: *host*."

- `cd` using the automounter

  This is a method of automatically mounting a file system when issuing a `cd` into it. It fast-fails with the following indication:

  "No such file or directory"

- `cd` into an NFS mounted file system

  The file system has already been mounted over the network (using the automounter or explicitly via `mount`). `cd` to the root of the mount point succeeds, because it requires no network traffic. All that is needed is a valid file handle. However, `cd` past the root of the mount point does require network traffic, and it fast-fails with these error messages.

  "NFS getattr failed for server <remote_host>: RPC: Program unavailable"

  "<target_file_system>: Network is down"

- `cat` NFS mounted file

  Immediate exit displaying:

  "cat: cannot open *file*"

- `cd` using OPENLOOK's `filemgr` into an NFS mounted file system

  Status display shows:

  "Unable to read directory for folder '*folder*': Network is down"

- edit an NFS mounted file with OPENLOOK's `textedit`

  Attempting to open an NFS mounted file brings up an error window:

  "The file `foo' does not exist. Please confirm creation of new file for textedit."

  Creating the new file exits the program with the following message:

  "NFS getattr failed for server <server>"

  "Unable to Save Current File. Cannot back-up file."

- ROAM nomadic mailtool

  Immediate notification in the status window:

  "Could not contact mail server"

  ROAM does not even display the login window. When instructed to connect, ROAM suggests the user try another mail server, as it has failed in trying to contact the default one. ROAM's disconnected mode is still available, though, and the user can compose messages, and queue them in the outbox.

7.2 System Isolated after Issuing the Command

Unless otherwise specified, the programs below exit immediately after the TCP connection is aborted. The usual behavior of exiting after a time-out is also available as a user-selectable option. The tests below were run with TCP aborts enabled.

- `rsh`, `rlogin`

  "Read error from network: Network is down"

  "Connection closed."

- `telnet`

  ```
  "Connection closed by foreign host"
  ```
- `ftp`

  `ftp` does not exit. Instead, it resets the connection, displays an error message:

  ```
  "421 Service not available, remote server
  has closed connection"
  ```

  and returns to the `ftp>` prompt.
- `rcp`

  ```
  "rcp: lost connection"
  ```
- `finger`

  The program displays only the information that made it before the network was severed. Since there is no error indication, this may be misleading.
- `spray`

  ```
  "SPRAYPROC_GET: RPC: Unable to send;
  System error"
  ```
- `rusers`

  ```
  "RPC: Unable to send"
  ```
- WWW browsers `Mosaic` and `netscape`

  If the http connection is severed during a transfer, there is no error indication. `netscape` even prints a completion status. The partial document is misleadingly displayed without any warnings. If the system becomes isolated after the http request has been sent but before the browsers receive the entire response from the server, there is no indication of failure. The browser waits until interrupted by the user.
- cd using OPENLOOK `filemgr` into an NFS mounted file system

  The status indicates that the

  ```
  "Network is down"
  ```

  Furthermore, the target directory's contents are displayed, but there is visual indication that not all the information was fetched. For example, if the file types are unresolved, their icons flag this condition.
- edit an NFS mounted file with OPENLOOK's `textedit`

  This can display a large number of error messages, specially when loading a file or scrolling. textedit can sometimes dump core and terminate. Simple operations like saving or reading simply generate a pop up error window:
- ROAM nomadic mailtool

  Immediate notification in the status window:

  ```
  "Connection to server broken"
  ```

  ROAM then enters its disconnected mode.

8.0 Programming Interface

This prototype provides isolation at the IP layer. An application can access IP either through a socket descriptor or a TLI file descriptor. In either case the application can receive the system isolation error through the standard error reporting facility supported by the transport layer.

In general, when an application sends or transmits a packet to the network, the library or system call completes successfully at the stream head, and then the data is forwarded down the stream. Thus, the application's transmit operation and the kernel's error reporting operation are asynchronous events. The next time the application accesses the socket or TLI file descriptor, the isolated condition is noticed and the system or library call exits with an appropriate error.

In most cases however, an application receives a system isolation indication prior to attempting a data transfer, because it typically uses NIS or DNS for name resolution. These detect the isolation and return an appropriate error to the calling program.

New applications can take advantage of the query ioctl discussed previously to ascertain whether or not the system is connected prior to initiating network activity.

Of course, if the application uses ICMP directly (ping, router discovery, etc.), it receives the ICMP packet generated to flag the isolation condition:

ICMP Type:	Destination Unreachable
Code:	8 (Source Host Isolated)

Applications that use UDP or TCP encounter the interfaces described below.

8.1 TLI

In the examples below, notice that the error is first noticed by some function setting `t_errno` to TLOOK, as is expected for all such asynchronous events [STEVENS 90].

Nevertheless, we could return TSYSERR and set errno to ENETDOWN. However, TLI already defines the TLOOK (event requires attention) mechanism to handle asynchronous events. Not doing so would depart from the interface specifications.

8.1.1 TCP case

System in isolated state before TCP connection is established.

The application issues `t_connect` to obtain a connection:

`t_connect:`

return value:	-1
t_errno:	TLOOK
errno:	0 (no error)

A subsequent call to `t_look` interrogates and clears the error condition as follows:

`t_look:`

return value:	T_DISCONNECT

TCP connection established before the system enters isolated state.

The application is already connected, so it sends data via `t_snd`, and succeeds. When TCP receives the subsequent ICMP host isolated error notification, it kills the connection. Upon reading the TLI endpoint via `t_rcv`, the error is noticed and the function call exits as follows:

`t_rcv:`

return value:	-1
t_errno:	TLOOK
errno:	0 (no error)

A subsequent call to `t_look` interrogates and clears the error condition as follows:

`t_look:`

return value:	T_DISCONNECT

If instead of reading or receiving, the application attempts to transmit again (`t_snd`) no error is reported. Errors are noticed when receiving

8.1.2 UDP case

TLI's connectionless send function is `t_sndudata`. If the system is isolated, this causes the ICMP error notification to propagate back from IP through UDP and the Transport Interface cooperating STREAMS module, `timod(7)`. When the TLI file descriptor is accessed with a connectionless receive (`t_rcvudata`) the error is noticed:

`t_rcvudata:`

return value:	-1
t_errno:	TLOOK
errno:	0 (no error)

`t_look:`

return value:	0
t_errno:	T_UDERR (datagram error indication)

`t_rcvuderr:`

return value:	0
unit data error:	ENETDOWN (Network is down)

If instead of reading or receiving, the application attempts to transmit again (`t_sndudata`) no error is reported. Errors are noticed when receiving.

The TLI endpoint remains usable. This is different from the current sockets interface.

8.2 Sockets

As explained previously, the network isolation error is encountered after the send operation has completed successfully. The sockets interface returns such asynchronous errors in subsequent operations on the socket. Alternatively, the SO_ERROR option of `getsockopt` can be used to interrogate the error [STEVENS 90]. Datagram sockets must be connected in order to receive such error reports. Furthermore, these are only available via `getsockopt`, but not as a result of socket I/O calls (`recv`, `recvfrom`, `send`, `write`, `read`, etc). In practice, this SO_ERROR interface is not very useful because applications rarely call `getsockopt` before accessing the socket endpoint.

Instead, in the UDP case, we decided that accesses to the socket endpoint return with the error condition. We do so by:

1. setting so_error to the error reported in the T_UDERROR_IND message (ENETDOWN), and
2. sending an M_ERROR message (with

ENETDOWN being the error on both read and write operations) up to the stream head.

The issue here is that once an M_ERROR is seen at the stream head, that stream is rendered unusable (i.e. the sockets interface is destructive). This is the main difference with the TLI interface. The rules for pipes, FIFO's and sockets in this situation are very clear: if a write operation is attempted on an unwriteable descriptor, SIGPIPE is generated [STEVENS 90]. It is the application's responsibility to catch this signal (see also the man pages for `socket`(3N) and `write`(2)).

It is also possible to avoid sending an M_ERROR to the stream head in response to a host isolated indication. Setting the sockets error variable so_error to the error ENETDOWN does allow the applications to fetch this information by using the SO_ERROR option to getsockopt. However, this does not allow current unmodified applications to fast-fail on network operations.

Furthermore, there have been discussions about allowing the user to enable and disable isolation semantics on a per-socket basis. This would allow finer granularity in deciding which applications fast-fail. It also guarantees that unsuspecting applications do not malfunction in the presence of fast-fail. The interface to this socket option would be something like:

- Add a new setsockopt option at the SOL_SOCKET layer named SO_FF_ENABLED.
- If SO_FF_ENABLED option is set ON, the transport layer will disconnect upon receiving host isolated errors. TCP will abort the connection, and UDP will permanently error out the socket.
- If the SO_FF_ENABLED option is set to OFF, the transport layer behaves exactly as it does today (no disconnection semantics).

The default behavior would be to not fast-fail, implying that current applications would not behave any differently unless explicitly modified. Because of this, we decided not to implement this interface in our prototype.

The interface seen by sockets programmers is as follows:

8.2.1 TCP case

System in isolated state before TCP connection is established.

The application issues `connect` to obtain a connection:

`connect`:

return value:	-1
errno:	ENETDOWN (Network is down)

TCP connection established before the system enters isolated state.

The application is already connected, so it sends data via `send`, `sendto`, `sendmsg` or `write`, and succeeds. When TCP receives the subsequent ICMP host isolated error notification, it kills the connection. Upon reading the socket endpoint via `read`, the error is noticed and the function call exits as follows:

`read`, `recv`, `recvfrom` or `recvmsg`:

return value:	-1
errno:	ENETDOWN (Network is down)

If instead of reading or receiving, the application attempts another `write` operation (or `send`, `sendto`, `sendmsg`), the result is:

(another) `write`, `send`, `sendto` or `sendmsg`:

return value:	-1
errno:	EPIPE (Broken pipe)
signal generated:	SIGPIPE

By default, SIGPIPE terminates the calling program.

8.2.2 UDP case

There is no information to match unconnected UDP sockets with incoming error notifications. Accordingly, it is not desirable to fast-fail applications that use unconnected UDP sockets.

Connecting a datagram socket involves nothing more than local caching of some information. Since there is no exchange with the target system, this step always succeeds. Applications transmit data on a datagram socket via any of the `write`, `writev`, `send`, `sendto`, or `sendmsg` functions. Even if the system is isolated these will succeed. However, this causes the ICMP error notification to propagate back from IP through UDP and the sockets kernel module. The error is seen when the socket is accessed via any of `recv`, `recvfrom`, `recvmsg` or `read`.

`read`, `recv`, `recvfrom` or `recvmsg`:

return value:	-1
errno:	ENETDOWN (Network is down)

If instead of reading or receiving, the application attempts another `write` operation (or `send`, `sendto`, `sendmsg`), the result is:

(another) `write`, `send`, `sendto` or `sendmsg`:

return value:	-1
errno:	EPIPE (Broken pipe)
signal generated:	SIGPIPE

By default, SIGPIPE terminates the calling program. The DNS resolver library caused this problem, and would terminate the calling program. To avoid this we modified the resolver as outlined in section 9.

The SO_ERROR option to `getsockopt` behaves as follows:

`getsockopt`(SO_ERROR):

return value:	-1
errno:	ENETDOWN (Network is down)

Again, the function itself fails (returns "-1"), because the M_ERROR message is destructive to the stream head.

We experimented with another variation of UDP socket isolation that avoided sending an M_ERROR to the stream head. In this case, we obtained the following interface:

"non-destructive" `getsockopt`(SO_ERROR):

return value:	0
so_error:	ENETDOWN (Network is down)
errno:	not applicable

This is the expected behavior of `getsockopt`: the value of so_error is returned successfully. Of course, no other function calls (i.e. `write`, `read`, `send`, `recv`) detect the isolation indication, and in order to allow current unmodified applications to fast-fail, we opted for the M_ERROR message.

9.0 Modifications to Network Services and Utilities

In addition to making changes to allow the error to propagate back up the protocol stack we had to make changes to the network services modules and some utilities so that they would react properly (i.e. fast-fail) to system isolation.

As mentioned above, sockets may fast-fail by sending an M_ERROR message up the stream head. This had the unpleasant effect of causing DNS queries to receive SIGPIPE with the default effect of terminating the calling program without any error message. We modified DNS to check for the isolation status of the system before attempting network operations. If the system is isolated, an error is returned instead of attempting the network access. This avoids the SIGPIPE problem. Currently, the check is done by using the newly defined isolation status query ioctl for IP. A better way might be to subscribe via the status subscription ioctl. DNS would then know the exact status without having to constantly poll for it. A more general solution would be to allow M_ERROR messages to be interpreted as non-persistent conditions. The next operation would both fetch and clear the error condition. The socket would not be destroyed, and it would remain usable.

RPC exists as both a kernel module for use by NFS and as part of a library. We modified both instances of RPC to handle correctly the unit data error indication received from the underlying layer, and to propagate the error RPC_CANTSEND to its clients. Handling this error correctly at the kernel RPC module enables NFS to fast-fail. Handling it correctly at the RPC library enables RPC applications (e.g. NIS) to fast-fail.

There are two cases to take care of in NIS:

System in isolated state before the NIS domain has been bound.

In this case, `ypbind` forks a child to broadcast for an NIS server. We decided not to allow the child to be spawned if the system was isolated. Accordingly, we query the network status in `ypbind` using one of the ioctl commands for IP. If the host is found isolated, a return of YPBIND_ERR_ERR causes NIS to fast-fail.

System in isolated state after the NIS domain has been *bound.*

> We modified the ypbind program to recognize the RPC error RPC_CANTSEND, and return the NIS error YPBIND_ERR_ERR.

The `mount` program receives RPC_CANTSEND as a result of isolated operation, and quickly terminates without retrying.

The `talk` program did not fast-fail when initiating a new session while isolated, because it used a disconnected UDP socket. Since fast-fail only applies to connected UDP sockets, `talk` never received the ENETDOWN error indication. This was fixed by connecting the control socket used by `talk`.

10.0 Implementation Details

Our prototype primarily consists of modifications to Solaris 2.4, Sun's SVR4-based operating system. Since the TCP/IP implementation is STREAMS-based, we had to modify various networking modules as follows (with the number of additional lines of code in parentheses):

- IP kernel module, `ip` (724)
- UDP kernel module, `udp` (169)
- TCP kernel module, `tcp` (163)
- sockets kernel module, `sockmod` (199)
- Transport Interface cooperating STREAMS module, `timod` (9)
- RPC kernel module, `rpcmod` (15)

Furthermore, we modified the following user-level code:

- `ping` program (8)
- `snoop` program (22)
- `talk` program (3)
- router discovery demon, `in.rdisc` (63)
- DNS client library, `libresolv.so` (120)
- Network Services Library, `libnsl.so` (57)
- `ypbind` program (227)
- NFS `mount` program (53)

Administrative programs and scripts represent an additional 695 lines of code.

11.0 Conclusion

We have developed a prototype that allows systems usually connected to the network to continue to function in a diminished capacity if the network is disconnected. The prototype uses an ICMP code to provide a fast-fail capability that allows unmodified applications to fail quickly when a network connection is not available.

Our framework allows for isolation aware applications and kernel modules to proactively handle system isolation by registering for and receiving notification as soon as system conditions change.

Acknowledgements

We thank the third member of the team, Becky Wong, for her help throughout the project. In particular, she modified the NIS, NFS and the `mount` clients to fast-fail upon receiving the notifications sent by IP. Erik Nordmark and Bob Gilligan provided many valuable comments and suggestions.

References

[WALDO 94] Jim Waldo, Geoff Wyant, Ann Wollrath and Sam Kendall; A Note on Distributed Computing, Sun Microsystems Laboratories Technical Report #TR-94-29, November 1994.

[PERKINS 95] Charlie Perkins, ed., IP Mobility Support, Internet Draft, January 1995.

[POSTEL 81] Jon Postel, RFC-792, *Internet Control Message Protocol (ICMP)*, September 1981.

[PERLMAN 92] Radia Perlman, *Interconnections: Bridges and Routers*, Addison-Wesley, 1992.

[BRADEN 89a] Robert Braden, RFC-1122, *Requirements for Internet hosts-communication layers*, October 1989.

[BRADEN 89b] Robert Braden, RFC-1127, *Perspective on the Host Requirements RFCs*, October 1989.

[ALMQUIST 94] Philip Almquist and Frank Kastenholz, RFC-1716, *Towards Requirements for IP Routers*, November 1994.

[STEVENS 90] W.Richard Stevens, *UNIX Network Programming*, Prentice-Hall, 1990.

[STEVENS 94] W.Richard Stevens, *TCP/IP Illustrated: the protocols*, Addison-Wesley, 1994

[POSTEL 94] Jon Postel, private communication, Message-Id: <199411120056.AA21711@zephyr.isi.edu>, November 11, 1994.

[CACERES 94] Ramon Caceres and Liviu Iftode. *The Effects of Mobility on Reliable Transport Protocols*. In Proceedings of the 14th International Conference on Distributed Computing Systems, June 1994.

A Generic Multicast Transport Service to Support Disconnected Operation*

Silvano Maffeis
maffeis@acm.org

Walter Bischofberger
bischi@ubilab.ubs.ch

Kai-Uwe Mätzel
maetzel@ubilab.ubs.ch

Department of Computer Science, Cornell University, and UBILAB, Union Bank of Switzerland

Abstract

Many mobile computing applications can profit from process groups and reliable multicast communication to maintain replicated data, but most operating systems available today fail in providing the primitive operations needed by such applications. In this paper we describe a highly configurable, Generic Multicast Transport Service (GTS), which supports the implementation of group-based applications in wide-area settings. GTS is unique in that it offers fault-tolerant, order-preserving multicast on arbitrary communication protocols, including e-mail. As another distinguishing mark, messages can be sent to processes even when they are temporarily unavailable, which permits disconnected operation and mobility. We further propose an object-oriented system design consisting of *adaptor objects* interconnected to form a *protocol tree*. Adaptor objects offer a common interface to dissimilar communication protocols, and make it easy to incorporate new protocols into GTS. Currently, GTS is being used in a cooperative software engineering environment and in other projects. GTS is available for anonymous ftp.

Keywords: Disconnected Operation, Replication, Message Spooling, Reliable Multicast, Wide Area Networks

1 Introduction

1.1 Motivation

Groupware, replicated file archives, and other kinds of distributed systems stimulate the need for structuring activities around process groups [3, 15] and reliable, order-preserving multicast [8]. We have developed a novel communication substrate, called the *Generic Multicast Transport Service (GTS)*, which enables the implementation of process group-based applications in wide-area networks. As the main abstractions, GTS offers reliable order-preserving multicast, reliable point-to-point communication, and process groups. A variety of transport protocols are supported and new protocols can be incorporated into the service easily. GTS is unique in that it permits disconnected operation, encrypted communication, reconfiguration, and in that applications may transmit messages without waiting until they have been delivered.

In the GTS system model, processes become unresponsive due to failures or due to disconnected operation of portable equipment. In such situations, GTS will spool messages on non-volatile storage and deliver them to their recipients as soon as they become available and register with GTS again. Addressing is by *Uniform Resource Locators* similar to the World Wide Web.

1.2 Related Work

Examples of state-of-the-art toolkits offering process groups and reliable, order-preserving multicast are CONSUL [12], ELECTRA [10], HORUS [14], ISIS [4], and TRANSIS [1]. These toolkits primarily aim to support applications running within one LAN. If a process has been unresponsive for a certain (usually short) period of time, their default behavior is to regard the process as faulty and to exclude it from the system. If a process or a whole group becomes unresponsive, applications cannot submit messages to it any more. Thus, periodic communication and disconnected operation are not adequately supported.

In contrast, GTS tolerates arbitrary communication delays and permits the sending of messages to disconnected processes, both by unicast[1] and multicast communication. In GTS, processes are never

*Research supported by grants from the Swiss National Science Foundation, Siemens-Nixdorf, Union Bank of Switzerland, and KWF/CERS

[1]point-to-point communication

excluded from the system unless explicitly requested by the user. Electronic retail banking, cooperative software engineering, software update protocols, distributed document servers, WAN applications to capture seismic signals, and replicated file archives are examples of applications which fit the GTS model. Moreover, GTS does not compete with the aforementioned toolkits but can be used in conjunction with them. Simply put, our scheme is well-suited for applications where disconnected operation, configurability, and widely distributed resources are more important than high-performance communication.

The ISIS wide-area facility [11] is the system most similar to GTS. At the heart of the facility lies a fault-tolerant application spooler which is restricted to a single LAN. ISIS applications can log messages into the spooler. An application that has failed will restart by initiating a spool replay operation, causing messages in the spooler to be played back to the application. During communication failures, messages are directed into a second spooler, called the *interLAN* area, and delivered to the destination after communication is re-established. The ISIS wide area facility supports process groups as well as totally ordered and causally ordered multicast. The main difference to GTS is that the ISIS wide-area facility is built on TCP, whereas GTS can run on virtually any communication protocol. Further, GTS embodies a flexible system design which permits to add functionality such as message compression or encryption easily. Last but not least, the wide-area facility is part of the ISIS toolkit whereas GTS is a stand-alone system. An important advantage of the ISIS facility is that it can be replicated over a LAN to increase availability. In contrast, a GTS LAN spooler is not replicated. However, availability can be increased by disk mirroring and by instantiating more than one GTS server process per LAN.

1.3 Organization of the Paper

The rest of the paper is structured as follows. Section 2 describes the system model, Uniform Resource Locators, and other important base concepts of GTS. GTS' system design is addressed in Section 3. In Section 4 the programming interface is presented along with simple example programs. Application experience with a cooperative software engineering environment based on GTS is reported in Section 5. Finally, Section 6 summarizes and concludes the paper.

2 Generic Multicast Transport Service

2.1 System Model

In our system model we distinguish between two kinds of processes: on one side are the GTS *servers*, which implement message spooling, reliable multicast, and unicast communication. On the other side stand the end-user *applications*, which use GTS. End-user applications can run on both mobile (laptops, palmtops, message pads) and immobile (hosts, workstations, PCs) equipment. A GTS server, along with the applications connected to it, makes up what we call a *cluster* (Figure 1). Typically, a cluster is contained in one LAN or within one mobile component. If an application in cluster A wants to send a message to an application in cluster B, it submits it to its server S_A, which in turn sends it to server S_B. Finally, S_B delivers the message to the end-user application.

GTS permits reliable unicast and multicast communication even when the underlying communication protocol is unreliable. In multicast communication, a sender application submits a message to a *group* of receiver applications. GTS guarantees that all members of the group receive the message, and that all members deliver messages in exactly the same order, which is called *totally ordered multicast* [8]. Multicast communication is useful for distributing the same data from one sender to a group of receivers efficiently, to perform computations redundantly, or to synchronize replicated data. GTS also supports groups of process groups. In WAN settings, this is useful for structuring large groups as a hierarchy of separately maintained subgroups.

A message to a disconnected destination is retained in the spooler of the GTS server which observes the disconnection. As was mentioned before, a mobile computer contains both a message spooler (i.e., a GTS server) and one or several applications. The server will try to deliver the message until it succeeds or until the destination is removed permanently from GTS. As a message travels through GTS, there will always be one copy of it in some spooler, and the server holding the copy is responsible for delivering it to the destination server or to the end-user application itself, if it runs in the server's cluster. If a GTS server fails, then nothing is lost since messages, group membership lists, and other important data are persistent.

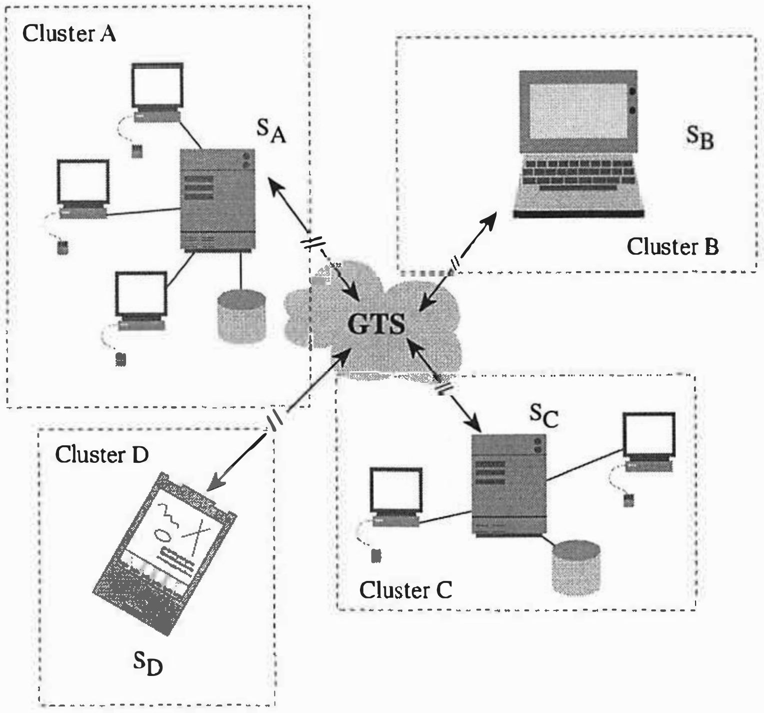


Figure 1: A typical GTS system configuration. S_i denotes the GTS server for cluster i. Applications are running on the workstations connected to their servers (Cluster A, C), on laptop computers (Cluster B), or on personal digital assistants (Cluster D).

2.2 Uniform Resource Locators

GTS supports an unrestricted set of protocols, for instance TCP, IP, AppleTalk, Mach Messages, ATM, e-mail, and UUCP. The API which programmers are confronted with is independent of the underlying transport protocols, and reliable multicast interprocess communication is feasible even with e-mail as the transport medium. In consequence, the addressing mechanism has to be simple and flexible. We decided to adopt the *Uniform Resource Locator (URL)* scheme proposed by the Internet Engineering Task Force. GTS URLs obey the following general form:
`protocol://cluster:server:localAddress/ticket`
The following GTS URLs all denote the same destination application:

```
tcp://ifi.unizh.ch:claude:9999/myApp
modem://ifi.unizh.ch:claude:(41)(1)3023570/myApp
uucp://ifi.unizh.ch:claude:Uclaude/myApp
email://ifi.unizh.ch:claude:gts/myApp
```

The first part of a URL defines the protocol used to deliver the message to its destination. The second part contains the address of the destination cluster. The `server` part contains the name of the GTS server. Hence, several GTS servers can run within the same cluster, if required. The `localAddress` is an internal, protocol-dependent address for the GTS server daemon, e.g. a TCP port number, a phone number, an e-mail account, and so forth. The `ticket` denotes the local application or application group the message is directed to. Here a character string stands for an endpoint application, a number for a group.

2.3 Privacy

GTS can be configured such that messages are transparently encrypted and decrypted using a public-key cryptosystem. Therefore, GTS maintains a public-key/private-key pair per URL. Typically, a GTS server provides the public keys of all destination URLs its client applications will send messages to, and it also maintains the private keys of its clients, such that incoming messages can be decrypted. If required, communication between a GTS server and its clients can be encrypted as well. Message encryption is accomplished by third-party software such as RSAREF or *Pretty Good Privacy (PGP)*. GTS' flexible system design (Section 3) permits easy incorporation of such encryption libraries.

2.4 Reliable Multicast Protocol

The multicast protocol we employ is similar to the one implemented in the AMOEBA [9] operating system. For each process group, there is one GTS server distinguished as the sequencer of the group. The sequencer maintains the URLs of the group members, and requests for joining or leaving the group must be directed to its sequencer.

To submit a multicast, the source server first delivers the message point-to-point to the sequencer server of the group (Figure 2). By inspecting the ticket of the destination URL, the server identifies the message as a multicast request, assigns the next multicast sequence number to it, looks up the URLs of the group members in a local membership file, and delivers the message to the member servers. Delivery is by one separate message per group member if the underlying transport protocol does not support multicast, or by one message for the whole group if all members can be reached with the same protocol, and given that the protocol supports multicast (e.g., IP with multicast extensions [2]). Note that a server can be sequencer and member of a group at the same time, and that member applications can be contained in the same cluster.

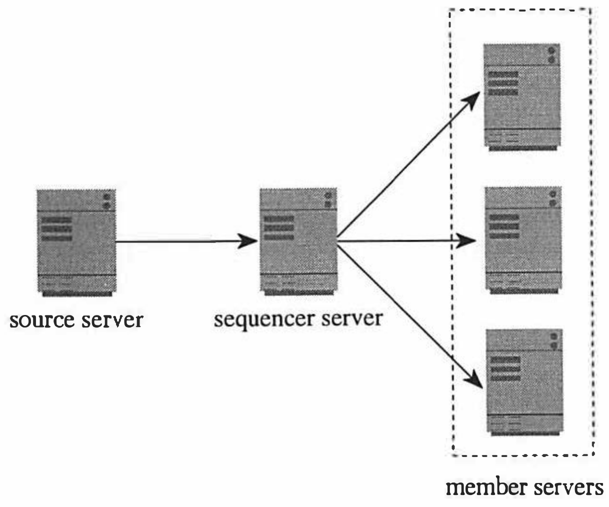


Figure 2: Message flow in the GTS multicast protocol.

Reliable multicast means that the members of a group agree on the set of multicasts they receive. If no damage to a source, sequencer, or member spooler occurs[2], GTS guarantees that a multicast is eventually received by all group members, and that members agree on the order of the multicasts they receive.

[2]e.g., due to a head-crash or a human lapse like accidentally deleting a spooler

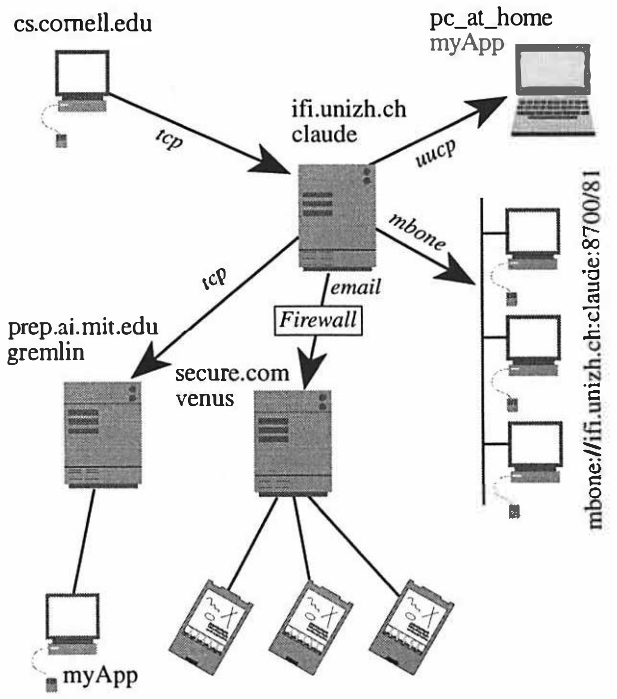


Figure 3: An example situation involving cascaded groups in a heterogeneous environment.

Consider the situation in Figure 3. Here, an application in the cluster `cs.cornell.edu` sends a message to the group
`tcp://ifi.unizh.ch:claude:9999/77.`
First, the sender's server delivers the message point-to-point to server `claude`. Since the ticket is a numeric value, `claude` retrieves the membership file for group 77 from its spooler. Assume that the file contains the entries
`tcp://prep.ai.mit.edu:gremlin:9999/myApp`,
`uucp://ifi.unizh.ch:claude:pc_at_home/myApp`,
`email://secure.com:venus:gts/31`, and
`mbone://ifi.unizh.ch:claude:8700/81`.
The message is multiplexed by server `claude` and forwarded to the four destinations. Since the third destination is a group maintained by the server of `secure.com`, which is reachable only by e-mail, the message is multiplexed by `venus` and transmitted to the message pads. The fourth destination also is a group; it contains members that can be reached by the IP multicast protocol [6] (also referred to as MBONE [2]).

3 Design of GTS

3.1 Protocol Tree

Several design goals guided the development of GTS:

- to support a wide range of protocols and operating systems,
- to make it easy for programmers to incorporate as yet unsupported protocols,
- to allow programmers to include their own API, and
- to devise a flexible design which other people can apply to their own systems.

This section focusses on the design of the GTS server, which is implemented in the C++ programming language. A GTS server is structured in a way similar to the x-kernel [13]. Each GTS server consists of a collection of *adaptor objects* plugged together to form a *protocol tree* as depicted in Figure 4. The root

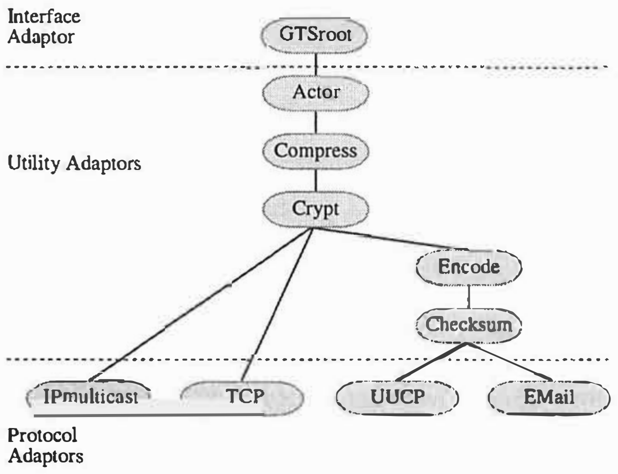


Figure 4: A sample protocol tree.

object (`GTSroot`) communicates with the client applications running in its cluster. Leaf objects, called protocol adaptors, perform unreliable message passing with remote servers by specific communication protocols. The utility adaptors in the middle area carry out tasks such as spooling and retransmission (the `Actor` object), compression, encryption, encoding, integrity check, and so forth in an operating system-independent fashion. Each adaptor object passes the messages it receives down the tree to one of its child adaptors. A message is routed through the tree according to the protocol part of its destination URL until it reaches a protocol adaptor. Finally, the protocol adaptor transmits the message to the destination server by the protocol it encapsulates.

At the destination server, the message is received by a protocol adaptor and is passed up the tree. If needed, it is checked, decoded, decrypted, and decompressed by the utility adaptors. The `Actor` adaptor spools the message to permit retransmission in case the end-user application is not available. Finally, the received message arrives at the `GTSroot` object where it is transmitted to the end-user application. Adaptor objects are organized in the form of the inheritance hierarchy depicted in Figure 5. Owing to this flexible system design, more than 90% of GTS' program code could be realized in a protocol- and operating system-independent way.

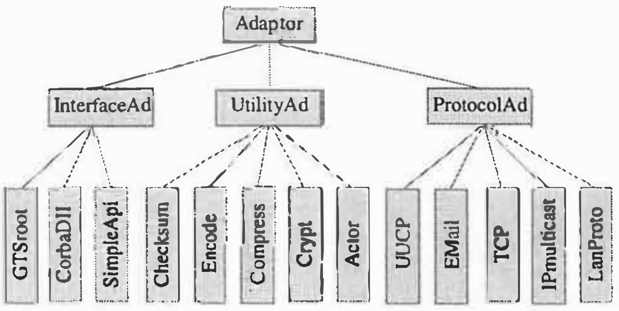


Figure 5: Adaptor inheritance hierarchy.

3.2 Adaptor Interface

A GTS adaptor object obeys the following interface:

```
class Adaptor {
public:
    virtual boolean down(Message&);
    virtual boolean up(Message&);
    virtual boolean done(Message&);
    virtual boolean viewChange(Message&);
    virtual boolean flush();
    virtual Adaptor& attach(Adaptor&);
};
```

An adaptor's `down` method is invoked by its father adaptor to pass down a message. To pass up a message, an adaptor invokes its father's `up` method. If an adaptor wants to discard a message (because it is corrupted or because it was received by the end-user application) it invokes the `done` method of all its child adaptors to ensure that they discard copies of the message they might hold. To force an adaptor to pass down the messages it stores, its `flush` method is invoked. The `viewChange` method informs an adaptor that an application joined or left a local group. Finally, the `attach` method is used to attach child adaptors and thus to construct a protocol tree.

For example, consider an adaptor to compress messages. Its down method compresses the data in the message, whereas its up method uncompresses the data. done, viewChange, flush, and attach need not be overwritten, thus the default behavior implemented in class Adaptor is inherited.

4 Using GTS

4.1 Programming Interface

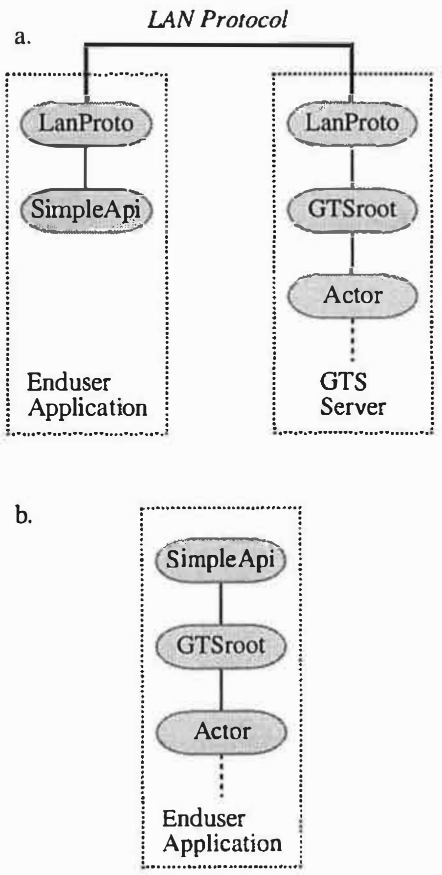


Figure 6: Coupling end-user applications with their GTS server.

In a LAN, end-user applications are linked with a communication stub that governs interaction with the cluster's server (with the GTSroot object, more exactly). This stub consists of an interface adaptor (SimpleApi) connected to a protocol adaptor (Figure 6 a.). The interface adaptor serves as an API to the programmer, whereas the protocol adaptor is used to communicate with the GTS server by a reliable LAN protocol like TCP or AppleTalk. In case GTS is installed on a mobile computer, end-user applications use the same interface adaptor as above, but can be directly linked with the server if required (Figure 6 b.). The present version of GTS provides the API below. As future work we plan to implement an API compatible with the CORBA *Dynamic Invocation Interface* [7].

```
class SimpleApi: public InterfaceAd {
public:
    // non-blocking send:
    boolean send(URL destination, const Message&);

    // blocking receive:
    boolean receive(OUT Message&,
        URL source =anybody);
    // non-blocking receive (polling):
    boolean receive(OUT Message&,
        OUT boolean& dataReady,
        URL source =anybody);

    // create a local URL group:
    boolean groupCreate(OUT unsigned int& groupID);
    // destroy a local URL group:
    boolean groupDestroy(unsigned int groupID);
    // join a URL group:
    boolean groupJoin(URL group, URL member);
    // leave a URL group:
    boolean groupLeave(URL group, URL member);
    // obtain the members of a local group:
    boolean groupGetInfo(URL group,
        OUT List& members);

    // obtain the URL of an application:
    boolean getMyURL(OUT URL& me);
};
```

The send operation submits a message to a destination URL without blocking the sender. To be suspended until a message has arrived, applications issue the first version of receive. To wait for a message from a specific sender, source is set to the URL of that sender. Otherwise the first arriving message is returned, regardless of the sender. The second version of receive is used to check whether a message is available without being blocked. If a message is available, it is assigned to the Message parameter and dataReady is set to TRUE. groupCreate/groupDestroy are used to create/destroy a group on the local server, groupID holds the group ID. groupJoin is used to add a member to a group, groupLeave to remove a member. groupGetInfo returns a list containing the URLs of the members of group. Finally, getMyURL serves to obtain the URL of an application.

4.2 Configuration

Various parameters of the GTS server can be tailored in a configuration file. For instance, permission to join or leave a group may be explicitly granted only to applications running in a certain cluster:

```
# ACCESS RIGHTS FOR LOCAL GROUPS:
#
# everybody can join or leave group 1:
group 1     jl      *
# only applications in cluster foo.edu can join group 2:
group 2     j       foo.edu
```

The return address assigned to an outgoing message can be controlled as well. This is useful for tailoring a GTS installation to environments where a firewall allows TCP traffic only in one direction, for instance.

```
# RULES TO SET THE RETURN ADDRESSES OF
# OUTGOING MESSAGES:
#
# The filters below are applied in sequence to the
# destination URL of an outgoing message until a
# match occurs.
#
# We request that replies to messages sent to bitnet
# or smallfirm.com destinations be returned to us by
# email:
myurl   *.bitnet        email
myurl   smallfirm.com   email
# uucp destinations shall submit replies by uucp:
myurl   *.uucp          uucp
# for the rest of the world we choose TCP:
myurl   *               tcp
```

For a cluster called `secure.com` residing behind a firewall (Figure 3), the `myurl` entries could be set as follows:

```
myurl    *secure.com    tcp
myurl    *              email
```

This means that within the `secure.com` domain, messages are exchanged by TCP. Replies for messages delivered to applications outside the domain will automatically arrive by e-mail to bypass the firewall. Thus, messages will be delivered by TCP in one direction and by e-mail in the other direction. For security, GTS can be set up such that messages crossing domain boundaries are protected by public key encryption (Section 2.3).

Protocol adaptors are configured as follows:

```
# CONFIGURATION OF PROTOCOL ADAPTORS:
#
protocol tcp    200000 30   300   9999 unicast
protocol email  100000 3600 28800 gts  unicast
protocol mbone  2048   30   300   9999 multicast tcp
```

In the above protocol configuration, the second entry assigns a name to the protocol adaptor. Messages traveling through the adaptor are split into fragments whose maximum size is controlled with the third entry. The fourth entry gives the initial retransmission interval in seconds, whereas the fifth entry the maximum interval. A retransmission interval is constantly increased until it reaches the maximum value. The sixth entry defines an address the adaptor uses to check for messages, for instance a TCP port number or an e-mail account. The seventh entry declares whether the underlying protocol supports multicast. In case of a multicast protocol, a further entry is supplied to specify by which protocol failed messages are retransmitted. For instance, when a multicast is sent through the `mbone` adaptor it is transmitted to the group by IP multicast [6, 2]. If the message fails to arrive at some destinations, it is transmitted point-to-point to them by `tcp`. If an adaptor does not support multicast, the `Actor` adaptor (Section 3.1) transparently multiplexes a multicast to one message per group member.

4.3 Examples

In the following example, a client application sends a request message to a weathermap server application, and then is suspended until a reply has arrived:

```
// client application:
//
SimpleApi gts;
...
Message request, reply;
request << "send weathermap of Detroit";

// submit the request:
gts.send("tcp://arc.nasa.gov:explorer:9999/mapServer",
    request);

// wait for the reply from the map server:
gts.receive(reply,
    "tcp://arc.nasa.gov:explorer:9999/mapServer");

// process received data ...
```

```
// server application:
//
SimpleApi gts;
...
Message request, reply;
// wait for a request message from any source:
gts.receive(request);

// process the request ...
// return a reply to the sender:
gts.send(request.getTrueFrom(), reply);
```

If the request is to be transmitted by e-mail, it is sufficient to change the above destination URL to `email://arc.nasa.gov:explorer:gts/mapServer`

In the next example, an application creates a group on its local GTS server and sends a multicast to it:

```
SimpleApi gts;
...
// variable to hold the group ID:
unsigned int gid;

// get the URL of this application:
URL group;
gts.getMyURL(group);

// create a group on the local server.
// Its group ID is assigned to gid:
gts.groupCreate(gid);

// modify the ticket of my URL to obtain
// the URL of the group:
group.setTicket(gid);

Message msg;
msg << "Hello World";

// join members to the group:
gts.groupJoin(group,
    "tcp://prep.ai.mit.edu:gremlin:9999/myApp");
gts.groupJoin(group,
    "uucp://ifi.unizh.ch:claude:pc_at_home/myApp");
gts.groupJoin(group,
    "email://secure.com:venus:gts/31");
gts.groupJoin(group,
    "mbone://ifi.unizh.ch:claude:8700/81");

// submit a multicast:
gts.send(group, msg);
```

5 Application Experience

At the UBILAB we are currently developing Beyond-Sniff, a platform with tools to support cooperative software engineering [5]. For developers connected by networks with high communication bandwidth (i.e. in LANs), cooperation is made possible by a number of distributed infrastructure services which rely on Beyond-Sniff's own mechanisms for data and control integration. Cooperation support over networks with low bandwidth and with portable destinations which are only temporarily active is based on a replication mechanism.

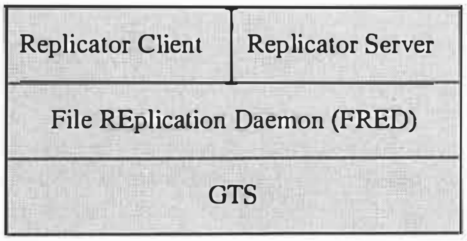


Figure 7: Architecture of the Beyond-Sniff Replicator.

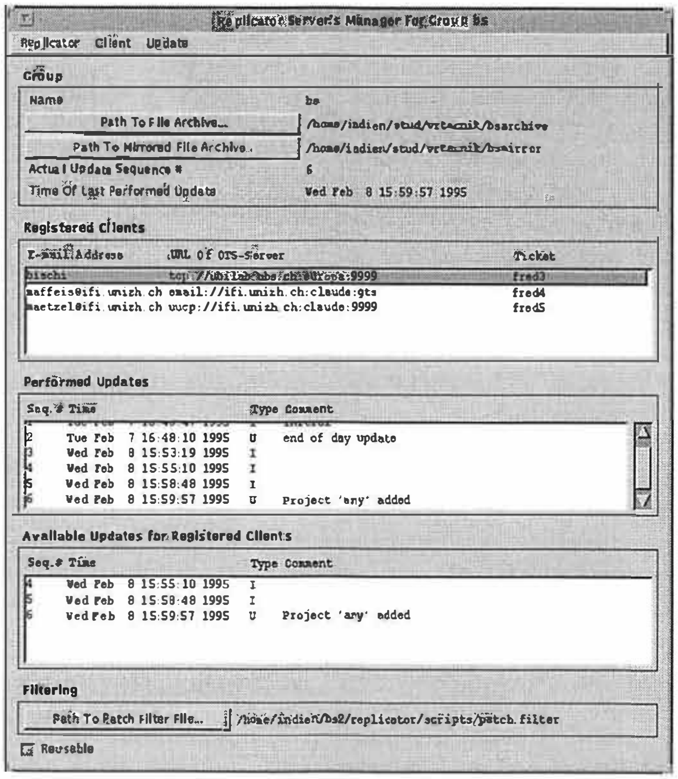
In the mid-term we intend to integrate the replication mechanism into Beyond-Sniff's infrastructure services such that it will become as transparent as possible to developers. To gain hands-on experience on how a replication-based approach influences software development in widely distributed teams, we have implemented and successfully applied the Replicator, a stand-alone replication mechanism based on GTS. The purpose of the Replicator is to reliably synchronize any number of remote client directories with a master directory tree. Its architecture consists of three levels as depicted in Figure 7.

The lowest level is formed by GTS which guarantees reliable data transfer between the master and the client sites over various protocols and in spite of temporarily disconnected clients. The GTS File Replication Daemon (Fred) creates, packages, transmits, and applies incremental updates to a replicated directory tree. Fred provides a simple command-line interface and can be used either as a background daemon, synchronizing automatically at well-defined time intervals, or as a slave synchronizing on demand. The highest level is formed by the Replicator's graphical user interface which consists of a server (Figure 8) and a client part (Figure 9). The interface mainly serves to let users easily access Fred's functionality.

From the user's perspective, the Replicator works as follows. To set up a synchronization group, the administrator of the master copy defines the directory to be mirrored and a further directory needed to calculate incremental updates. The administrator then defines which clients will be supported by specifying their URLs as well as the personal e-mail addresses of the persons in charge of administering the clients. When a new client joins a replication group, a complete version of the directory hierarchy is packaged and sent to it. When a directory is updated, the packaging and multicasting of updates to the clients is triggered explicitly on the master site.

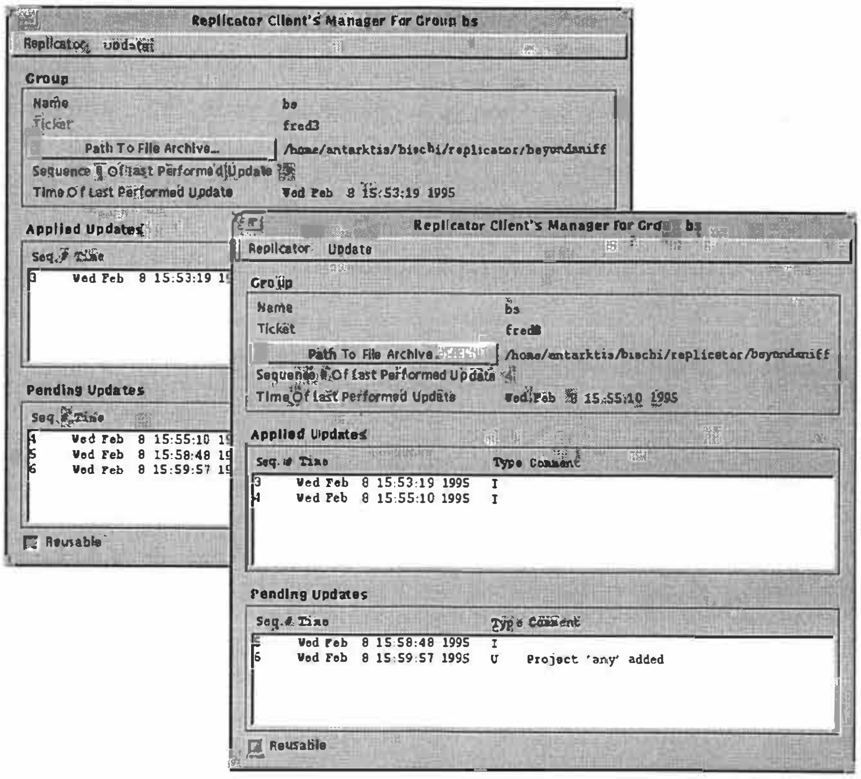
The client administrators are automatically notified by an e-mail message when a data transfer took place. The graphical user interface on the client side (Figure 9) mainly presents lists with new and already applied updates. Updates are explicitly applied when it makes sense in the context of the ongoing cooperation.

The implementation of Fred and of the Replicator's user interface was straightforward (about 10 working days) and it has proven the usefulness of GTS as well as the adequacy of its API. Without GTS we would not have been able to develop the Replicator in a reasonable time. The Replicator is successfully being applied to synchronize our joint development efforts between sites in Switzerland, Germany, and Austria. We intend to make the Replicator publicly available in the near future.


Replicator Server's Manager For Group bs
Replicator Client Update
Path To File Archive...
Path To Mirrored File Archive...
Path To Patch Filter File...
Reusable

```
Group
Name                                  bs
Path To File Archive...               /home/indien/stud/vrtacnik/bsarchive
Path To Mirrored File Archive...      /home/indien/stud/vrtacnik/bsmirror
Actual Update Sequence #              6
Time Of Last Performed Update         Wed Feb  8 15:59:57 1995

Registered Clients
E-mail Address          URL Of OTS-Server                         Ticket
bischi                  tcp://ubilab.ubs.ch:8Urora:9999           fred3
maffeis@ifi.unizh.ch    email://ifi.unizh.ch:claude:gts           fred4
maetzel@ifi.unizh.ch    uucp://ifi.unizh.ch:claude:9999           fred5

Performed Updates
Seq.# Time                        Type Comment
2     Tue Feb  7 16:48:10 1995    U    end of day update
3     Wed Feb  8 15:53:19 1995    I
4     Wed Feb  8 15:55:10 1995    I
5     Wed Feb  8 15:58:48 1995    I
6     Wed Feb  8 15:59:57 1995    U    Project 'any' added

Available Updates for Registered Clients
Seq.# Time                        Type Comment
4     Wed Feb  8 15:55:10 1995    I
5     Wed Feb  8 15:58:48 1995    I
6     Wed Feb  8 15:59:57 1995    U    Project 'any' added

Filtering
Path To Patch Filter File...          /home/indien/bs2/replicator/scripts/patch.filter
```

Figure 8: User interface of the Replicator server.



```
Group
Name                                  bs
Ticket                                fred3
Path To File Archive...               /home/antarktis/bischi/replicator/beyondsniff
Sequence # Of Last Performed Update   3
Time Of Last Performed Update         Wed Feb  8 15:53:19 1995

Applied Updates
Seq.# Time
3     Wed Feb  8 15:53:19 19

Pending Updates
Seq.# Time
4     Wed Feb  8 15:55:10 19
5     Wed Feb  8 15:58:48 19
6     Wed Feb  8 15:59:57 19
```

```
Group
Name                                  bs
Ticket                                fred3
Path To File Archive...               /home/antarktis/bischi/replicator/beyondsniff
Sequence # Of Last Performed Update   4
Time Of Last Performed Update         Wed Feb  8 15:55:10 1995

Applied Updates
Seq.# Time                        Type Comment
3     Wed Feb  8 15:53:19 1995    I
4     Wed Feb  8 15:55:10 1995    I

Pending Updates
Seq.# Time                        Type Comment
5     Wed Feb  8 15:58:48 1995    I
6     Wed Feb  8 15:59:57 1995    U    Project 'any' added
```

Figure 9: User interface of the Replicator client.

6 Conclusions

Widely-distributed systems often need to synchronize replicated data in spite of disconnected equipment and failures. In this paper we presented a novel communication substrate, called the *Generic Multicast Transport Service (GTS)*, which was developed at the University of Zurich and at the Union Bank of Switzerland. The development of GTS was influenced by the results of projects such as AMOEBA, ELECTRA, ISIS, and *x*-kernel. GTS is different from previous work on process group-based systems mainly in that it deals with disconnected operation and in that it focusses on widely-distributed rather than on local resources. Moreover, a flexible, object-oriented system design consisting of *adaptor objects* interconnected to a *protocol tree* has been devised. This system design permits reliable multicasts to be issued on arbitrary transport protocols, for example on TCP/IP or even on e-mail, if necessary, and messages are addressed to Uniform Resource Locators. New functionality can be added to GTS easily by developing new plug-in adaptors.

Presently, GTS is being used to build heterogeneous distributed applications interconnecting several clusters. As an example of a real-world application employing GTS we described Beyond-Sniff, a cooperative software engineering environment. In our experience, GTS is ideal for replicating data in a distributed system consisting of static and mobile computing equipment. We also found that groupware serving asynchronous forms of collaboration often requires the kind of system support this paper proposes.

As future work we plan to port GTS to PC operating systems and to personal digital assistants. We also intend to incorporate GTS into a CORBA event channel service such that widely distributed CORBA objects can communicate through GTS.

Availability

GTS is available for anonymous ftp in the directory `ftp://ftp.ifi.unizh.ch/pub/projects/gts/`. Information on Beyond-Sniff and on the Replicator can be retrieved from `ftp://ftp.ubilab.ubs.ch`. Universities can obtain a free copy of the non-distributed SNiFF+ programming environment from `ftp://self.stanford.edu/pub/sniff/`.

References

[1] AMIR, Y., DOLEV, D., KRAMER, S., AND MALKI, D. Transis: A Communication Sub-System for High Availability. In *22nd International Symposium on Fault-Tolerant Computing* (July 1992), IEEE.

[2] BAKER, S. Multicasting for Sound and Video. *Unix Review* (Feb. 1994).

[3] BIRMAN, K. P. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM 36*, 12 (Dec. 1993).

[4] BIRMAN, K. P., AND VAN RENESSE, R., Eds. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, 1994.

[5] BISCHOFBERGER, W. R., KOFLER, T., MÄTZEL, K.-U., AND SCHÄFFER, B. Computer Supported Cooperative Software Engineering with Beyond-Sniff. In *Proceedings of the 7th Conference on Software Engineering Environments* (Noorwijkerhout, The Netherlands, 1995).

[6] DEERING, S. Host Extensions for IP Multicasting. RFC 1112, Request for Comments, Aug. 1989.

[7] DIGITAL EQUIPMENT CORP., HEWLETT-PACKARD CO., HYPERDESK CORP., NCR CORP., OBJECT DESIGN INC., SUNSOFT INC. *The Common Object Request Broker: Architecture and Specification*, Dec. 1993. Revision 1.2.

[8] HADZILACOS, V., AND TOUEG, S. Fault-Tolerant Broadcasts and Related Problems. In *Distributed Systems*, S. Mullender, Ed., second ed. Addison Wesley, 1993, ch. 5.

[9] KAASHOEK, M. F., TANENBAUM, A. S., HUMMEL, S. F., AND BAL, H. E. An Efficient Reliable Broadcast Protocol. *ACM SIGOPS Operating Systems Review 23*, 4 (Oct. 1989).

[10] MAFFEIS, S. A Flexible System Design to Support Object-Groups and Object-Oriented Distributed Programming. In *Proceedings of the ECOOP '93 Workshop on Object-Based Distributed Programming* (1994), R. Guerraoui, O. Nierstrasz, M. Riveill, Ed., Lecture Notes in Computer Science 791, Springer-Verlag.

[11] MAKPANGOU, M., AND BIRMAN, K. Designing Application Software in Wide Area Network Settings. Tech. Rep. 90-1165, Department of Computer Science, Cornell University, Oct. 1990.

[12] MISHRA, S., PETERSON, L. L., AND SCHLICHTING, R. D. Consul: A Communication Substrate

for Fault-Tolerant Distributed Programs. *Distributed Systems Engineering Journal 1*, 2 (Dec. 1993).

[13] PETERSON, L., HUTCHINSON, N., O'MALLEY, S., AND RAO, H. The x-kernel: A Platform for Accessing Internet Resources. *IEEE Computer 23*, 5 (May 1990).

[14] VAN RENESSE, R., AND BIRMAN, K. P. Fault-Tolerant Programming using Process Groups. In *Distributed Open Systems*, F. Brazier and D. Johansen, Eds. IEEE Computer Society Press, 1994.

[15] VERÍSSIMO, P., AND RODRIGUES, L. Group Orientation: A Paradigm for Distributed Systems of the Nineties. In *Proceedings of the Third Workshop on Future Trends of Distributed Computing Systems* (Apr. 1992), IEEE Computer Society.

Partially Connected Operation

L.B. Huston
http://www.citi.umich.edu/u/lhuston

P. Honeyman
http://www.citi.umich.edu/u/honey

Center for Information Technology Integration
University of Michigan
Ann Arbor

ABSTRACT

RPC latencies and other network-related delays can frustrate mobile users of a distributed file system. Disconnected operation helps, but fails to use networking opportunities to their full advantage. In this paper we describe *partially connected operation*, an extension of disconnected operation that resolves cache misses and preserves client cache consistency, but does not incur the write latencies of a fully connected client. Benchmarks of partially connected mode over a slow network indicate overall system performance comparable to fully connected operation over Ethernet.

1. Introduction

An important advantage of a distributed computing environment is on-demand access to distributed data. Disconnected operation [11, 7], a form of optimistic replication that allows access to cached data when file servers are unavailable, has proved successful at providing this access to mobile users. Disconnected operation is especially successful at hiding network deficiencies by deferring and logging all mutating operations, replaying them later.

Distributed systems tend to be designed to work in environments that provide high data rates and low latencies, but these assumptions are generally invalid in a mobile environment. Here, disconnected operation has broad applicability, but is something of a blunt instrument: because it treats networks as either available or unavailable, disconnected operation does not account for the varying degrees of network quality encountered by mobile users.

For example, even though AFS [6] caches aggressively and has good support for low-speed networking in the transport protocol [1], the network latency that accompanies many operations can make AFS over a low-speed network a trying experience. This affects user satisfaction when interactive response time is increased beyond that which a user is willing to tolerate.

One option is to use disconnected operation when only a low bandwidth network is available, using the network solely to satisfy cache misses. This approach does not support the AFS cache coherence mechanisms, so a user may unwittingly use stale data at a time when it is possible to obtain the most recent version. Furthermore, mutating operations are not propagated immediately, increasing the chance that two users might concurrently update the same file.

Lying between connected and disconnected operation is a mode of operation that allows us to hide many of the network latencies, but to continue to use the network to maintain a relaxed form of cache consistency. In the remainder of this paper, we give an overview of our approach and some implementation details, and present some benchmarks that illustrate the effectiveness of the technique.

2. Background

The work presented in this paper is based on a version of the AFS client that supports disconnected operation [7]. The client cache manager supports three modes of operation; connected, disconnected, and fetch-only. In connected mode the cache manager is an ordinary AFS client, using callback promises to preserve cache coherence [10]. In disconnected mode the cache manger treats the network as unavailable, and allows cached data to be used even though cache consistency can not be guaranteed. File and directory modifications are also handled optimistically: updates are reflected in the disconnected cache and logged for later propagation to the file server when the decision is made to return to connected operation. Conflict due to overlapping updates while disconnected is possible, but rare.

Fetch-only mode is similar to disconnected mode, but differs in that it processes cache misses by requesting the needed data from the server. We use fetch-only mode frequently, both at home and when traveling, to bring missing files to a client without the cost of a full replay.

When a network is available, the user may choose to return to connected operation. The cache manger replays the log of deferred operations by iterating through the operations and propagating the modifications to the server. Before any operation is replayed, the cache manager examines server state to make sure someone else's newly created data is not destroyed. Manual error recovery is invoked if such a conflict occurs.

3. Related work

Our work with disconnected operation is inspired by the CODA project, which introduced the concept of disconnected operation and identified its usefulness for mobility [11]. CODA researchers are working on support for low bandwidth networks, such as predictive caching to obviate network demands caused by cache misses, and trickle discharging, which shares our goal of using network connectivity opportunistically without interfering with other traffic [3].

The Echo distributed file system is similar to ours in its use of write behind to reduce the latencies of operations and improve performance [14]. We depart from the Echo approach in two important ways. The first is failure semantics. We log synchronously, so when an operation completes, its changes are committed to the log and will eventually be replayed. Echo applications must either call `fsync` or a special operation that guarantees the order in which operations are committed to the server.

Echo enforces single system UNIX semantics by demanding delayed updates from client machines. In the mobile environment this requirement might be expensive or impossible to honor and can project the bandwidth latencies of mobile networks onto users of a high speed network.

4. Partially connected operation

We now describe *partially connected operation*, a technique for mobile systems that lies between connected and disconnected operation. As in disconnected operation, all file system writes are performed locally and logged. The main differences from disconnected operation are in the way it maintains client cache coherence and processes cache misses.

In partially connected mode, as in disconnected operation, `vnode` operations that cause file modifications are processed by modifying the file cache to reflect the update and creating a log entry. In some cases the ordinary AFS cache manager delegates error checking to the server, but we need to fail invalid operations as they occur, so we modified the cache manager to perform the necessary checks locally.

In disconnected mode, the cache manager behaves as though the network were unavailable and optimistically assumes that all cached data is valid. In contrast, partially connected mode assumes the availability of some communication between the client and file servers. This lets us use AFS callbacks to offer regular AFS consistency guarantees to the partially connected client: such a client opening a file is guaranteed to see the data stored when the latest (connected) writer closed the file [10]. A partially connected client sees local modifications to files before they have been propagated to the server, as do connected clients.

Directories can be tricky. A partially connected user may insert a file in a directory, while another user inserts another entry into the directory. If the cached version of the directory is used (because it has local modifications not yet propagated to the server), the entry inserted by the other user will not be seen, so we have to fetch a fresh copy of the directory and merge in our update. We plan to address this problem in the future.

On low bandwidth networks, the user may not always want the most recent version of files. For example if any files under `/usr/X11/bin/` are modified, the user may wish to continue using the cached versions instead of incurring the cost of fetching the most recent version.† We are investigating methods of providing an interface to allow this form of selective consistency.

5. Background replay

In disconnected operation, file modifications are not propagated immediately, making it difficult to share data consistently and increasing the likelihood of a conflict during replay [12]. For partially connected operation, we want to take advantage of network availability no matter what the quality if it lets us achieve timely propagation of updates, so we implemented a

† At some point the current version should be brought into the cache, but this can be deferred to a background task that runs when the system is otherwise quiescent.

background daemon to replay the log whenever opportunities arise or at the user's option.

Two significant issues arise when replaying operations in the background. First is rational management of network resources, so that the response times for interactive and other traffic do not suffer. The second issue is the effect on optimization: we and our CODA counterparts have observed that optimization of large logs can be considerable [8, 17], vastly reducing the amount of network traffic necessary for replay. Aggressive background replay may deny us this savings.

5.1. Priority queuing

The network is a primary resource in the mobile environment, so it is vital to keep replay traffic from interfering with a user's other work. Competition among various types of network traffic can increase interactive response time by causing interactive traffic to be queued behind replay traffic. Studies have shown that interactive response time is important to a user's satisfaction [18].

Similarly, replay traffic might compete with an AFS `fetch`, which is undesirable if a user is waiting for the completion of the associated read request. No user process blocks awaiting replay, so replay operations should be secondary to all other network requests.

One solution is to replay operations when the network is otherwise idle. In practice this solution is hard to implement; it is difficult to tell when a network (or other resource) is idle [4]. Furthermore, some operations, such as `store` requests, may take several minutes to complete. To avoid interference with interactive traffic, the replay daemon would need to predict a user's future behavior.

Our solution is to augment the priority queuing in our network driver. Our approach is an extension of Jacobson's compressed SLIP [9] implementation, which uses two levels of queuing in the SLIP driver: one for interactive traffic, and one for all other traffic. When the driver receives a packet for transmission, it examines the destination port to determine which queue to use. When ready to transmit a packet, it first transmits any packets on the interactive queue. The low priority queue is drained only when the interactive queue is empty.

We extend this approach by using three levels of queuing: interactive traffic, other network traffic, and replay traffic. AFS `fetch` requests are put on the second queue because a user is likely to be waiting for the completion of a read request.

When determining which packet to transmit we depart from the Jacobson method. In his SLIP implementation, the packet with the highest priority is always sent first, which for our purposes might lead to starvation of the low priority queue(s). For example, suppose the replay daemon is storing a file in the background and the user starts a large FTP PUT. FTP packets takes precedence over replay traffic, so no replay traffic will be transmitted during the duration of the FTP transfer. If the FTP transfer lasts long enough, the AFS connection will time out, and lose any progress it has made on the operation being replayed.

To prioritize the queues without causing starvation, we need a sophisticated scheduler that guarantees a minimum level of service to all traffic types. We use lottery scheduling, which offers probabilistic guarantees of fairness and service [19].

Lottery scheduling works by giving a number of lottery tickets to each item that wants to access a shared resource. When it is time to choose which item gets use of the resource, a drawing is held. The item holding the winning ticket gets access to the resource. This give a probabilistic division of the access to the resource based on the number of tickets that each item holds.

In our driver, we assign a number of tickets to each of the queues, according to the level of service deemed appropriate. When it is time to transmit a packet we hold a drawing to determine which queue to transmit from. Ticket allocation is a flexible way to configure the system and provides an easy-to-understand ''knob'' to turn for system tuning.

For the measurements described in this paper, we gave eight tickets to the interactive queue, three to the demand network traffic queue, and one to the replay queue. In future work, we plan to measure the effect of varying these static assignments. Under some circumstances, we may elect to vary them dynamically.

5.2. Delayed writes

Effective crash recovery is critical whenever delaying writes. We commit all file and metadata modifications to the log as quickly as possible, so that we don't have any dirty data in the buffer cache in a crash. Log replay works after a client crash — in our prototypes, we rely on it often (sadly).

Distinct connected clients that sequentially write a shared file don't experience a conflict, but delaying one of the updates can produce a concurrent write sharing conflict when replaying the log, so it is to our advantage to replay the log without much delay. In addition, delaying update for too long hampers the timeliness and potential usefulness of shared data; there is good reason to propagate logged operations aggressively.

On the other hand, delaying replay offers an opportunity for optimizing the log. Ousterhout reported that most UNIX files have a lifetime under three minutes and that 30–40% of modified file data is overwritten within three minutes [16]. Using our optimizer [8], we find it typical for 70% of the operations in a large log to be eliminated. It is clear that delaying log replay can significantly reduce the amount of data propagated to the server.

We may wish to enforce a minimum delay before replaying an operation, especially on networks with a per-packet cost, so that optimization could have an effect. On the other hand, if the network is available and idle and costs nothing to use, then there is nothing to be saved. On our dialups or our Ethernet, we propagate changes aggressively, whenever surplus network bandwidth is available. We run the optimizer only when changing from disconnected to connected or partially connected operation. In the future, we plan to experiment with different approaches to configuring the delay according to the varying network characteristics.

6. Results

To see how well partially connected operation performs in the low-speed and intermittent networks that interest us, we measure running times for several benchmarks. We start with hot data and attribute caches, so that comparisons between Ethernet and low-speed networks are meaningful. Later we examine the effect of a cold attribute cache.

We ran the benchmarks over Ethernet, over SLIP in connected mode (C-SLIP), and over SLIP in partially connected mode (P-SLIP). Measurements were made on a 33 Mhz Intel 486 client running Mach 2.5. We used SLIP on a 57.6 Kbps null modem connection to avoid the variability in network transfer time due to modem latencies and compression.

6.1. nhfsstone benchmark

To measure the fine-grained effect of partially connected operation on individual operations, we ran the `nhfsstone` benchmark [13], modified to remove NFS dependencies. The results in Table 1 show that P-SLIP runs substantially faster than C-SLIP, as we would expect.

Operations that run slowly over Ethernet run faster in P-SLIP, which has the advantage of deferring network requests. Because all P-SLIP operations involve synchronous logging, there is a lower bound to the running time for any particular operation.

Operation	Ethernet	C-SLIP	P-SLIP
`setattr`	21	516	33
`write`	255	3,517	123
`create`	218	2,036	99
`remove`	62	629	41
`rename`	23	294	41
`link`	36	319	40
`symlink`	125	383	47
`mkdir`	129	635	73
`rmdir`	132	351	40

Table 1. Comparison of mutating operation completion times. This table compares the time to perform various vnode operations in three cases: over an Ethernet, over SLIP in connected mode, and over SLIP in partially connected mode. The measurements were made using a version of `nhfsstone`. All times are in ms.

6.2. Andrew benchmark

We ran the Andrew benchmark [5], a synthetic workload that copies a file hierarchy, examines the copied files, and compiles source files. This benchmark, summarized in Table 2, shows that partially connected operation dramatically improves the running time of the Andrew benchmark: partially connected mode over SLIP is much faster than its connected counterpart. Because many network delays are removed from the critical path, the benchmark runs only a little slower on P-SLIP than over Ethernet.

	Ethernet	C-SLIP	P-SLIP
`MakeDir`	4	29	2
`Copy`	31	191	23
`ScanDir`	18	26	45
`ReadAll`	28	30	40
`Make`	117	507	116
`Total`	200	783	218

Table 2. Andrew benchmark results. This table shows the running time the Andrew benchmark over Ethernet in connected mode, and over SLIP in connected and partially connected mode. Both SLIP measurements started with hot data and attribute caches. All times are in seconds. Integer roundoff accounts for differences between column entries and column totals.

These examples show that partially connected mode improves the response time for file system operations. With a hot cache, a remote user can expect tasks to run almost as fast as in the office environment.

6.3. Replay time

We measured the log replay time for the Andrew benchmark. Figure 1 shows the size of the log as the Andrew benchmark runs through its phases. The solid horizontal line along the bottom of the graph shows the running times of the MakeDir, Copy, and Make phases of the benchmark. (The ScanDir and ReadAll phases are read-only.) The dashed horizontal line shows the time at which the corresponding parts of the log were replayed.

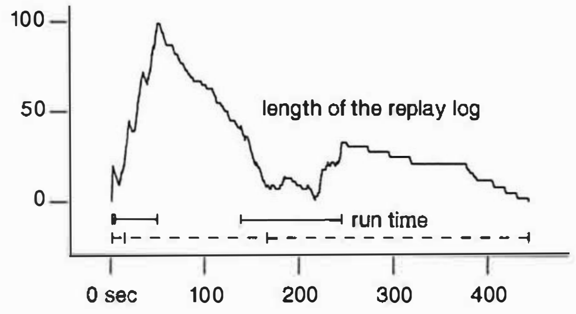


Figure 1. Length of replay log. This figure shows the number of operations in the replay log while running the Andrew benchmark over partially connected SLIP. The solid horizontal line show the running time of the MakeDir, Copy, and Make phases of the benchmark. The ScanDir and ReadAll phases are not shown, as they issue no network requests. The dashed horizontal line shows the times at which the operations logged by these three phases are run.

A total of 222 operations are logged, with no more than 99 operations pending at any time. Table 3 shows that logged operations are delayed over a minute on average, and up to four minutes in the extreme.

	average	maximum
cold cache	62	207
hot cache	87	249

Table 3. Delay time in the replay log. This table shows the average and maximum time that operations await replay while running the Andrew benchmark. When running with a cold attribute cache, the benchmark is frequently stalled, giving the replay daemon more opportunities to work on the log, resulting in shorter average and maximum delays. The total time to run the benchmark and exhaust the log is about the same in both cases. All times are in seconds.

6.4. Replay interference

The running times of the Andrew benchmark vary according to whether the data and attribute caches are hot or cold, as well as whether the replay daemon is running or idle. Table 4 shows the effect of controlling these variables.

	I	II	III	IV
cache	hot	cold	cold	hot
replay	off	off	on	on
MakeDir	3	3	2	3
Copy	19	26	47	23
ScanDir	14	15	48	45
ReadAll	25	25	40	40
Make	94	96	106	116
Total	155	165	249	218

Table 4. Cold cache Andrew benchmark results. This table shows the effect of running the Andrew benchmark with a hot or cold attribute cache, and with the replay daemon running or disabled. All times are in seconds. Integer roundoff accounts for differences between column entries and column totals.

A cold attribute cache slows the pace of the benchmark, giving the replay daemon more opportunities to whittle away at the log in the earlier phases, so that the average and maximum delay of logged operations is decreased, as shown in Table 3. The replay daemon itself causes the benchmark to slow down by 40–50% overall. This may be due in part to network contention, but even the ScanDir and ReadAll phases, which hit hot caches and don't use the network at all, run slower when background replay is active. This points the finger at contention for local resources; we suspect competition for locks in the AFS cache manager to be a contributor.

There is one mysterious entry in Table 4: the run time of the Copy phase for cold cache + active replay daemon (case III, 47 sec.). This is the hard case, where background replay has the most opportunities to interfere with interactive operations. Still, the penalty is higher than we expect, and we think there is a bug in our implementation.

To isolate the effect of network interference caused by running the replay daemon, we ran the Andrew benchmark in fetch-only mode with the replay daemon turned on and off. The run times should be comparable to the corresponding hot-cache trials in Table 4, but in fact the benchmark runs quite a bit faster in fetch-only mode, as shown in Table 5.

The difference in run-times caused by enabling the replay daemon in fetch-only mode is a little more than 10%; we believe that we will be able to debug and tune our implementation to achieve these differences in partially connected mode as well. With replay disabled, hot cache run times do not differ much for partially connected mode, fetch-only mode, and disconnected mode (not shown).

	I′	IV′	local
cache	hot	hot	——
replay	off	on	——
MakeDir	2	2	3
Copy	18	23	13
ScanDir	14	16	13
ReadAll	24	42	23
Make	93	104	88
Total	152	197	139

Table 5. Andrew benchmark results for fetch-only mode and local disk. This table shows the effect of running the Andrew benchmark in fetch-only mode with a hot attribute cache, and with the replay daemon running or disabled. All times are in seconds. Integer roundoff accounts for differences between column entries and column totals. For comparison, local disk times are also shown here.

7. Discussion

Partially connected operation promises to improve response time and reliability, while interfering only slightly with AFS cache consistency guarantees. AFS guarantees that a client opening a file sees the data stored when the most recent writer closed the file. Because a partially connected client does not immediately propagate changes, other users can not see modified data. Furthermore, conflicts may occur if partially connected users modify the same file. In our experience, these conflicts are rare; a substantial body of research concurs by showing that this kind of file sharing is rare [2, 11, 16].

If stronger guarantees are needed, they might be provided by server enhancements. For example, an enhanced consistency protocol might inform servers that dirty data is cached at a client; when another client requests the data, the server can demand the dirty data, as is done in Sprite [15] and Echo.

We choose not to implement this mechanism for several reasons. First, it assumes that the server is able to contact the client on demand, an assumption that may not always be true. Additionally, demand fetching can place a severe strain on a client's network connection. Because of the limited bandwidth, one user may see her effective bandwidth drastically reduced because another user is reading a file that she has modified; this may not be acceptable to all users. Finally, such a change would require changing all of the AFS servers in the world to support the new protocol; practically speaking, this is out of the question.

References

1. D. Bachmann, P. Honeyman, and L.B. Huston, "The Rx Hex," in *Proc. of the First Intl. Workshop on Services in Distributed and Networked Environments*, Prague (June 1994).
2. Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout, "Measurements of a Distributed File System," in *Proc. of the 13th ACM Symp. on Operating Systems Principles*, Asilomar (October 1991).
3. Maria R. Ebling, Lily B. Mummert, and David C. Steere, "Overcoming the Network Bottleneck in Mobile Computing," in *Proc. of the IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz (December 1994).
4. Richard Golding, Peter Bosch, Carl Staelin, Tim Sullivan, and John Wilkes, "Idleness is Not Sloth," pp. 201–212 in *Proc. of the USENIX Conf.*, New Orleans (January 1995).
5. J.H. Howard, M.L. Kazar, S.G. Menees, D.A. Nichols, M. Satyanarayanan, R.N. Sidebotham, and M.J. West, "Scale and Performance in a Distributed File System," *ACM Transactions on Computer Systems* **6**(1) (February, 1988).
6. John H. Howard, "An Overview of the Andrew File System," pp. 23–26 in *Proc. of the Winter USENIX Conf.*, Dallas (January 1988).
7. L.B. Huston and P. Honeyman, "Disconnected Operation for AFS," in *Proc. of the 1993 Symp. on Mobile and Location-Independent Computing*, Cambridge (August 1993).
8. L.B. Huston and P. Honeyman, "Peephole Log Optimization," in *Proc. of the IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz (December 1994).
9. V. Jacobson, "Compressing TCP/IP Headers for Low-Speed Serial Links," RFC 1145, Network Information Center, SRI International, Menlo Park (February 1990).
10. Michael Leon Kazar, "Synchronization and Caching Issues in the Andrew File System," in *Proc. of the Winter USENIX Conf.* (February 1988).
11. J.J. Kistler and M. Satyanarayanan, "Disconnected Operation in the Coda File System," *ACM Transactions of Computer Systems* **10**(1) (February 1992).
12. James J. Kistler, "Disconnected Operation in a Distributed File System," Ph.D. Thesis, Carnegie Mellon University (May 1993).
13. Legato Systems, Inc., *NHFSSTONE*, July, 1989.

14. T. Mann, A. Birrell, A. Hisgen, C. Jerian, and G. Swart, "A Coherent Distributed File Cache with Directory Write-behind," SRC Research Report #103, Digital Equipment Corporation (June 1993).

15. M. Nelson, B. Welch, and J. Ousterhout, "Caching in the Sprite Network File System," *IEEE Transactions on Computers* **6**(1) (February 1988).

16. J. Ousterhout, H.L. DaCosta, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson, "A Trace-Driven Analysis of the Unix 4.2 BSD File System," in *Proc. of the 10th ACM Symp. on Operating Systems Principles*, Orcas Island, WA (December 1985).

17. M. Satyanarayanan, James J. Kistler, Lily B. Mummert, Maria R. Ebling, Pumeet Kumar, and Qi Lu, "Experience with Disconnected Operation in a Mobile Computing Environment," in *Proc. of the 1993 Symp. on Mobile and Location-Independent Computing*, Cambridge (August 1993).

18. B. Shneiderman, *Designing the User Interface,* Addison-Wesley (1987).

19. Carl A. Waldspurger and William E. Weihl, "Lottery scheduling: flexible proportional-share resource management," pp. 1–11 in *Proc. First Symp. on Op. Sys. Design and Impl. (OSDI)*, Monterey (Nov. 1994).

Availability

Researchers with an armful of source licenses may contact `info@citi.umich.edu` to request access to our AFS modifications.

A Distributed Software Architecture For GPS-Driven Mobile Applications

Thomas G. Dennehy†
Environmental Research Institute of Michigan
Ann Arbor, MI 48113-4001

ABSTRACT

The unique requirements of voice recognition can shape a software architecture in many ways that have proven effective for mobile and distributed applications. We show in this paper that extending the voice recognition model of translating utterances into sentences to include translating a variety of real-world events into a command protocol can create an architecture whose components operate identically on hand-held devices, man-portable or vehicle-borne units, notebook, or desktop computers. SANSE, a portable navigation and geographic information management system having several redundant user interfaces, is described. In SANSE a collection of distributed *Interactors* translate events—spoken words, input from GPS hardware, timers expiring, input from files or communication links, and direct manipulation actions—into SANSE commands that are sent to one or more *Receivers*, which can execute commands without regard to their source. The complete operation of this system can be captured in vocabulary of less than 70 words, small enough to provide speaker-independent operation yet rich enough to be broadly applicable. The architecture can be extended by adding new Interactor types without affecting the operation of the baseline system.

1. Introduction

SANSE is a software architecture for GPS-driven mobile applications that developed from a simple yet challenging concept: to build a portable navigation and geographic information management system with two completely redundant user interfaces: direct manipulation and voice-activated. The unique requirements of voice recognition shaped the SANSE architecture in a number of ways that proved effective when configuring systems for stand-alone or networked operation. Components of SANSE-based systems can be deployed on hand-held devices, man-portable or vehicle-borne units, notebook, or desktop computers.

For a command language to be effective, it must satisfy a number of criteria [1]:

- *Expressiveness* – The language must provide complete access to the capabilities of the system.
- *Expressiveness of Intent* – The vocabulary must be precise enough, but the user should not be overburdened with expressing his intent. Commands should be short—a few words at most—and to the point.
- *Freedom from Detail* – The vocabulary should be interpreted within a general context in order to cut down the detail that needs to be expressed. This should not be confused with *context-sensitive* grammars, which allow a single word to be interpreted in multiple ways depending on the local sentence context. Such word overloading should be avoided in command languages.
- *Principle of Least Surprise* – The commands and vocabulary should be familiar and natural and behave in expected ways. A Geographic Information System (GIS) may have several related definitions of **North**, for example, and although a particular language may choose to recognize only one of these definitions, the language should not redefine **North** to mean what is generally recognized as **South**.

But an effective command language provides not only a convenient means to *use* a system, but also a natural structure around which to *organize* the system, a particularly effective structure for distributed systems. First, by defining system behavior in terms of a well-understood command set, we can effectively decouple the

†The author is currently an independent consultant based in Bloomfield Hills, MI. E-mail to `af510@detroit.freenet.org`.

response to a command from the various and often redundant circumstances that can initiate the command. Second, the command set defines the internal protocol of the system, creating abstract interfaces between architectural components so these components will interact identically whether deployed on a common platform or distributed across a hardware network. Finally, short commands make effective use of inter-process, packet, cellular, and other protocols.

In the next section, we describe the core SANSE architecture for a system with direct manipulation and voice activation, and how this model can be extended for a variety of other input sources. The SANSE protocol is then described, followed by a description of a SANSE-based portable navigation system and discussion of future directions.

2. Core System Architecture

Creating two redundant user interfaces, voice-activated and direct manipulation, is a difficult problem since these two interface styles communicate very differently.

Voice recognition hardware translates utterances into one of several forms, the most common of which are: *isolated words*, where a token representing each individual recognized word is returned to the host; and *connected speech*, where speech energy is interpreted according to a sentence grammar loaded onto the hardware, returning legal sentence structures. To accommodate both styles, SANSE chose sentences as the basis for the protocol between the **Voice Interactor** and the SANSE core. If an isolated word recognizer were chosen, it would be the responsibility of the Voice Interactor to assemble the words into legal sentences.

Visual interface toolkits have various styles of communication. User input can cause events to be posted (X Windows), messages to be sent (Microsoft Windows) or callback function to be invoked (Xt toolkit, Motif widgets). Complete redundancy between the two user interfaces required the **Screen Interactor** to relate to the SANSE core in the same way as the Voice Interactor, making the callback structure not feasible. While events or messages are a valid basis for communication, they didn't match the natural output of the Voice Interactor. The Screen Interactor was therefore designed to translate direct manipulation actions into sentences.

With outside world interfaces translating external events into sentences, the core SANSE component—the **Receiver**—was designed to accept and interpret the SANSE command protocol (Figure 1). From the Receiver's perspective, SANSE operation is a stream of commands that can be executed without regard to the circumstances that created them. From the user's perspective, there is no difference between speaking the words `PAN LEFT` or pressing the corresponding button on screen, allowing voice commands and direct actions to be freely mixed.

The Receiver executes a command by updating one or more state variables know as *Subjects*. Each Subject has one or more *Views* associated with it that needs to be notified when the Subject is modified. A View is owned by an Interactor, and is simply a representation—visible, audible, or hidden—of one or more Subjects[2]. This Subject/View coupling yields a simple deterministic model for implementing the command set; the complexity of the control structure of the system is independent of the number of commands recognized.

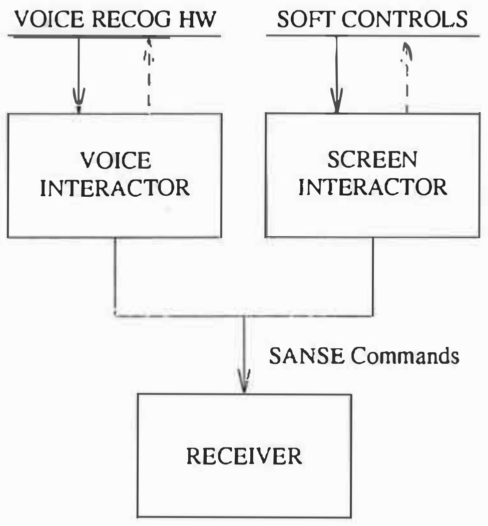


FIGURE 1. INTERACTORS TRANSLATE EXTERNAL EVENTS INTO COMMANDS IN THE SANSE PROTOCOL.

The operation of SANSE can be partitioned into three separable occurrences (Figure 2):

1) External events are handled by the various Interactors, which translate those events into SANSE commands and send the commands to the Receiver.
2) The Receiver executes commands, modifying one or more Subjects per command executed.
3) When a Subject is modified, it notifies the Views which are currently attached to it, so that the individual Views may update their appearance to reflect the new state of the Subject.

View update is a different process from the completely local graphic occurrences intended to provide interactive feedback. For example, when a "soft" button is pressed, its appearance will be altered so as to inform the user that the action has been registered—the shading of its borders may invert, for example—but that action reflects only that a particular button was pressed, not that a particular SANSE command was executed as a result.

To illustrate, an on-screen button may cause a PAN LEFT command to be sent when pressed, but saying

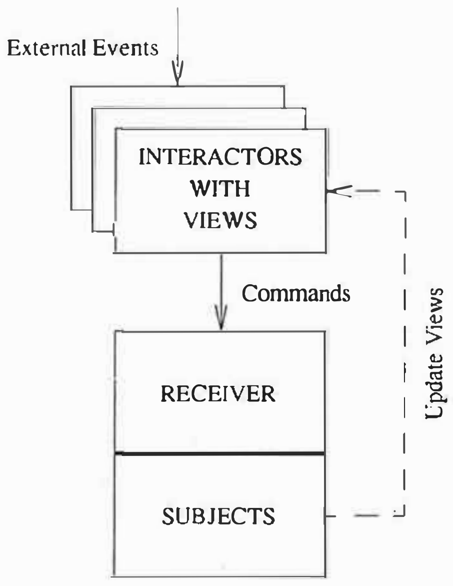


FIGURE 2. COMMAND EXECUTION IS A DETERMINISTIC PROCESS INDEPENDENT OF LANGUAGE SIZE.

"Pan Left" should not cause the same visual feedback, as though some invisible hand had pressed the button. However, panning left will update any View that is tied to the Subject representing the Forward direction, and that update will occur independent of whichever Interactor sent the command.

The Interactor model can be extended to any structure that translates physical events into SANSE commands to be interpreted by the Receiver. Three more Interactors have proven immediately useful:

- **GPS Interactor**, which translates real-time global positioning information into SANSE commands.
- **Trap Interactor**, which originates SANSE commands in response to elapsed time or distance traveled;
- **Remote Interactor**, which relays SANSE commands received over remote links or read from files.

The Interactors model the redundancy of the user interface—silent operation or hands-free operation—as well as the independence of the interface components. Given this redundancy and independence, different SANSE systems with various combinations of Interactors can be configured, and the extensibility of the system is well defined. New capabilities, may be added to the system without affecting its present operation by defining a new Interactor to translate new types of events into commands to send to the Receiver, extending the SANSE command vocabulary if necessary.

From the Receiver's perspective any SANSE Interactor is a drop-in replacement for any other Interactor, enabling consistent operation in both stand-alone and distributed configurations. For example, SANSE would operate identically as a self-contained portable navigation system receiving input from an on-board GPS receiver (through the GPS Interactor) or as a desktop tracking system receiving position information from one or more mobile systems (via a Remote Interactor). The Interactors and the Receiver communicate through an abstract interface that can be implemented using a variety of physical channels and protocols[3].

Because Interactors operate independent of one another and independent of the Receiver, SANSE systems can be deployed in stand-alone or networked configurations using a wide variety of hardware components.

- A simple field data collection application can be hosted on a hand-held device using only the GPS and Trap Interactors, operating either in batch mode or in real-time communication with a base station via radio or cellular links. (SANSE's command protocol is well-suited to new packet cellular protocols like CDPD.)
- Portable systems incorporating voice response and GIS displays have been hosted on notebook computers outfitted with single-board peripherals.
- Shadow systems (where mobile system A reports its position to desktop or mobile system B) have been deployed with both systems A & B having full display capabilities.

Advances in CPU power, PCMCIA packaging, and storage capacity will make such self-contained SANSE systems no larger or heavier than the notebook computers hosting the software.

3. The SANSE Protocol

SANSE's command protocol has two representations, an internal packet format. and an ASCII equivalent. The ASCII representation of a command is a sequence of fields separated by semi-colons and terminated by a newline.

```
Org;Mnemonic;C;Keyword;Data;T_Sent;T_Rec
```

The `Org` identifies the command as user-generated (`U`) or system-generated (`S`). Each command `Mnemonic` can have optional `Keyword` and/or `Data`. Data representations for geographic positions, headings, GPS status packets, and numeric choices have been devised; others can be easily added. The `T_Sent` (Time Sent) is supplied by the Interactor originating the command; the `T_Rec` (Time Received) is inserted by the Receiver.

There are two mechanisms for repeating command execution. SANSE will repeat once the last user command executed whenever the Receiver gets the `MORE` command, or will continuously repeat the last user command when the Receiver gets the `CONTINUE` command. The continuation process repeats until the next user command is received. System commands (new GPS location or status information, for example) can be executed without interrupting continuation.

The `C` (Continuation) field of the command protocol has proven valuable for cutting down the communication load between the Receiver and Interactors. Placing the string "ING" in the `C` field is a request for immediate continuation. Thus, if a button owned by the Screen Interactor is intended to provide sustained operation, it can send a command with the Continuation field set when the button is pressed, and send a `STOP` command when the button is released. The communication load is therefore independent of the amount of time the button is depressed, and the controls operate effectively in networked configurations. The Voice Interactor uses the present participle form of certain commands to request continuation: "Panning Left" as opposed to "Pan Left."

Thus, the ASCII representation of the user command `PAN LEFT` would be

```
U;PAN;;LEFT;;;
```

while `PAN LEFT CONTINUE` would appear as

```
U;PAN;;LEFT;;;
U;CONTINUE;;;;
```

but could be appreviated as

```
U;PAN;ING;LEFT;;;
```

Sample content for the data field is illustrated by a choice command like `USE 2`:

```
U;USE;;C 2;;
```

The internal representation of this protocol is fixed-length packets; the packet size is determined by the size of the largest data element it can contain, currently 20 bytes. The unused packet space in commands that contain only keywords or shorter data is more than compensated by avoiding the overhead of sending and receiving data-dependent variable-length packets. SANSE components residing on the same host almost always use the internal representation for routing commands; uncoupled components can choose the ASCII or internal format as required.

The Receiver maintains a history file of all commands executed, with commands stored in their ASCII representation. History files can be replayed through the Remote Interactor. During replay, the Remote Interactor can reproduce or accelerate the relative gaps between commands represented by their individual `T_Rec` stamps.

4. A SANSE Vocabulary for GPS/GIS Applications

Plates 1 and 2 following the text of this paper illustrate a SANSE-based portable system combining navigation and geographic information display with multimedia field data collection and review. This system was implemented with a command set of 33 operations and a total vocabulary of 70 words, a vocabulary small enough to provide speaker-independent voice response. This section describes the operation of that system and its vocabulary.

4.1 Perspective commands. These commands alter the *Field of View*, the region of the earth's surface represented by the GIS display.

Mnemonic	*Argument*
TRACK(ING)	Direction
PLACE	Location or KnownPosition
PAN(ING)	LEFT or RIGHT
LOOK	NumericHeading, Direction, or KnownPosition
TIGHTEN(ING)	
WIDEN(ING)	
ZOOM	IN or OUT
ENLARGE	
REDUCE	
CONVERGE	

The *View Point*—the center of the Field of View—is typically the position reported by the GPS Interactor, but can be established at an absolute location using the `PLACE` command, which takes as its argument a geographic Location or the keywords `HERE`, representing the location currently reported by the GPS Interactor, or `BACK`, representing a previously stored location (see section 4.4). The Field of View can be moved incrementally by `TRACK(ING)` in any of the four compass directions (`NORTH`, `SOUTH`, `EAST`, `WEST`) or `FORWARD`, `BACKWARD`, `LEFT`, or `RIGHT` relative to the *View Heading*. The View Heading is typically the current heading reported by the GPS Interactor, but can be rotated by `PAN(ING) LEFT` or `RIGHT` or positioned at an absolute heading using the `LOOK` command.

The extent of the Field of View (the *scale* of the display) can be changed using the `ENLARGE` or `REDUCE` commands. The degree of enlargement of reduction is controlled by the View Finder, whose size is controlled by the `TIGHTEN` and `WIDEN` commands. `ZOOM IN` makes the View Finder as small as it can be; `ZOOM OUT` removes it from the screen. Finally, the CONVERGE commands restores the display scale to the natural scale of the data being viewed.

4.2 Composition commands. These commands manipulate data sets shown in the Field of View;.

Mnemonic	*Argument*
USE	Choice or NONE
WITH	Choice or NONE
ADD	Choice or ALL
REMOVE	Choice or ALL
HIDE[a]	
SHOW	

a.Equivalent to REMOVE ALL.

The display model combines a raster-based underlay image with vector or symbol-based overlays (annotation). The underlay is a composition of two classes of data: *backgrounds* and *transparencies*. Backgrounds might be scanned maps or satellite photos, while transparencies include land use maps, elevation maps, or related data sets. Although a single background or single transparency could function as the underlay image, there are a number of background/transparency combinations that make tactical sense. The various categories of annotation are rank-ordered by priority, and the enabled overlays shall be drawn in reverse order of priority, lowest to highest.

The USE command specifies the background data set to use, or NONE. The equivalent command for the transparency is WITH. ADD and REMOVE manipulate layers of the overlay; HIDE turns off the current overlays; SHOW restores them. Background, transparency, and overlay choices can be always specified by number, and a number of common types of date (MAP, PHOTO, TRACE) have been assigned keywords in the vocabulary. Future versions of the system may support loading customized vocabularies to represent specific data sets.

4.3 Screen Management commands. These commands interact with the window management system through the Screen Interactor;.

Mnemonic	*Argument*
OPEN	Window
WHERE AM I	
CLOSE	Window
RAISE	Window
MOVE(ING)	Direction
BEFORE	
NEXT	

OPEN and CLOSE can be used to configure the display. The window types recognized are:

- VIEW – A window showing the Field of View, along with controls for opening other windows;
- SCALE – showing the current map scale, along with controls for changing scale and manipulating the View Finder;
- COMPASS – showing position, heading, and status information reported by the GPS Interactor;
- KEY – showing the current composition of the Field of View, along with controls for manipulating backgrounds, transparencies, and overlays.
- POINT – showing the View Point and View Heading, along with controls to manipulate them.
- MARKER – showing information about user-defined markers (see next section).

WHERE AM I is a natural equivalent to the command OPEN COMPASS.

Most windows are referenced by their keyword alone, but MARKER windows are referenced by name and number ("Open Marker 4") or the most recent Marker if the number is omitted. The user can chain through the entire list of Markers once a Marker window is open using the BEFORE and NEXT commands. This same feature could be extended to other kinds of windows representing data maintained in lists.

The RAISE command brings a particular window to the top and makes it the current window. Although a command is provided for MOVE(ING) the current window UP, DOWN, LEFT, or RIGHT, using this command is admittedly far less convenient than using a pointing device.

No attempt is made in this command subset to provide access to all the features of a particular window management system or toolkit.

4.4 Action commands. These commands report data, mark locations, and initiate other miscellaneous actions.

Mnemonic	*Argument*	*Qualifier*
MORE		
CONTINUE		
STOP		
REF		
UNREF		
MARK		SYSTEM or USER
GPS_FIX	GPS Fix	SYSTEM
GPS_STATUS	GPS Status	SYSTEM
CHECK		SYSTEM or USER
QUIT		SYSTEM or USER

As previously discussed, MORE repeats one the last user command executed. CONTINUE repeatedly executes the last user command until the next user command is received. STOP interrupts continuation without executing another command. The REF command saves the current View Point and View Heading; these values can then be accessed through the keyword BACK (as opposed to HERE). UNREF clears these values.

System-initiated commands are distinct from user commands in that system commands do not interrupt continuation, but instead have their execution interleaved with continuation. GPS_FIX and GPS_STATUS are System commands that relay position, heading, and status reports from a GPS receiver. The System command MARK is initiated by the Trap Interactor periodically to save the current GPS position and heading on a *Trace* of travel. The interval between Trace point can be time-based, distance-based, or a combination. The other System commands are CHECK, to run a self-test, or the self-evident QUIT; both these commands can also be user-initiated.

One other System command can also be user-initiated. A user can leave a MARK at the current View Point, with an option to annotate that mark with data exchanged with other programs. SANSE applications have used Markers containing spreadsheet data, CAD drawings, or audio recordings (an example of an audio Marker is shown in Plate 2). A Marker file with annotation can be preloaded into SANSE, enabling SANSE to be used in the field to update spatial database information in its native format.

5. Discussion

SANSE is written in the C++ language and used primarily on computers running the Microsoft Windows operating system (Release 3.1 and later). SANSE systems can exchange data with other programs through the Microsoft OLE protocol, with SANSE acting as the OLE client. SANSE can be ported to other operating environments, as it incorporates no proprietary nonstandard language features and processes only seven Windows messages in the course of its operation.

Although originally written for use in mobile GPS/GIS applications, the SANSE architecture provides a robust general model for mobile and distributed systems by: 1) defining system behavior in terms of a well-understood command set; 2) effectively decoupling the response to a command from the various and often redundant circumstances that can initiate the command; and 3) creating abstract interfaces between architectural components so these components will interact identically whether deployed on a common platform or distributed across a hardware network. This approach pays several benefits:

- Redundancy – In representing the redundancy between elements of the operator interface (silent operation and hands-free operation, for example) the architecture also models the independence of the various elements and how they individually relate to the SANSE core.
- Configurability – Since elements of the SANSE interface are independent, the architecture supports instantiating SANSE with elements selectively enabled/disabled. The command set is easily partitioned; a distributed system can have several Receivers, each recognizing only those commands that can make use of the local platform resources.
- Extensibility – In establishing the allocation of functionality between the SANSE core and various elements of the interface, the architecture supports extending the interface by adding new I/O devices—a video camera, for example—without affecting the operation of the present system.

To write distributed programs, one must be conversant in two distinct vocabularies. First there is the vocabulary of the problem domain, or the *computation model*; software embodying the computation model is called application code. Second, there is the vocabulary of the system domain, or the *coordination model*; software embodying the coordination model is called system code. It has been shown elsewhere that a well-chosen vocabulary for system code can isolate application code from the details of physical process distribution and communication channels, creating distributed programs that may be conveniently ported across different operating environments [3]. Here we have shown that a well-chosen application vocabulary extends this flexibility to the application components, creating systems whose core functionality is isolated from the many redundant sources of its inputs and outputs, and whose diverse components can serve as drop-in replacements for one another to serve a broad range of needs.

6. Acknowledgments and Contact Information

SANSE was designed at the Environmental Research Institute of Michigan (ERIM) in Ann Arbor, MI. The author wishes to acknowledge the many contributors to the project: Orest Mykolenko, Matt Frazer, Lori Sulik, and Linda Spencer for software design; Dave Symanow, Cyrus Wood, and Len Tomko for hardware design and logistics; and especially Ron Swonger for his vision and management support.

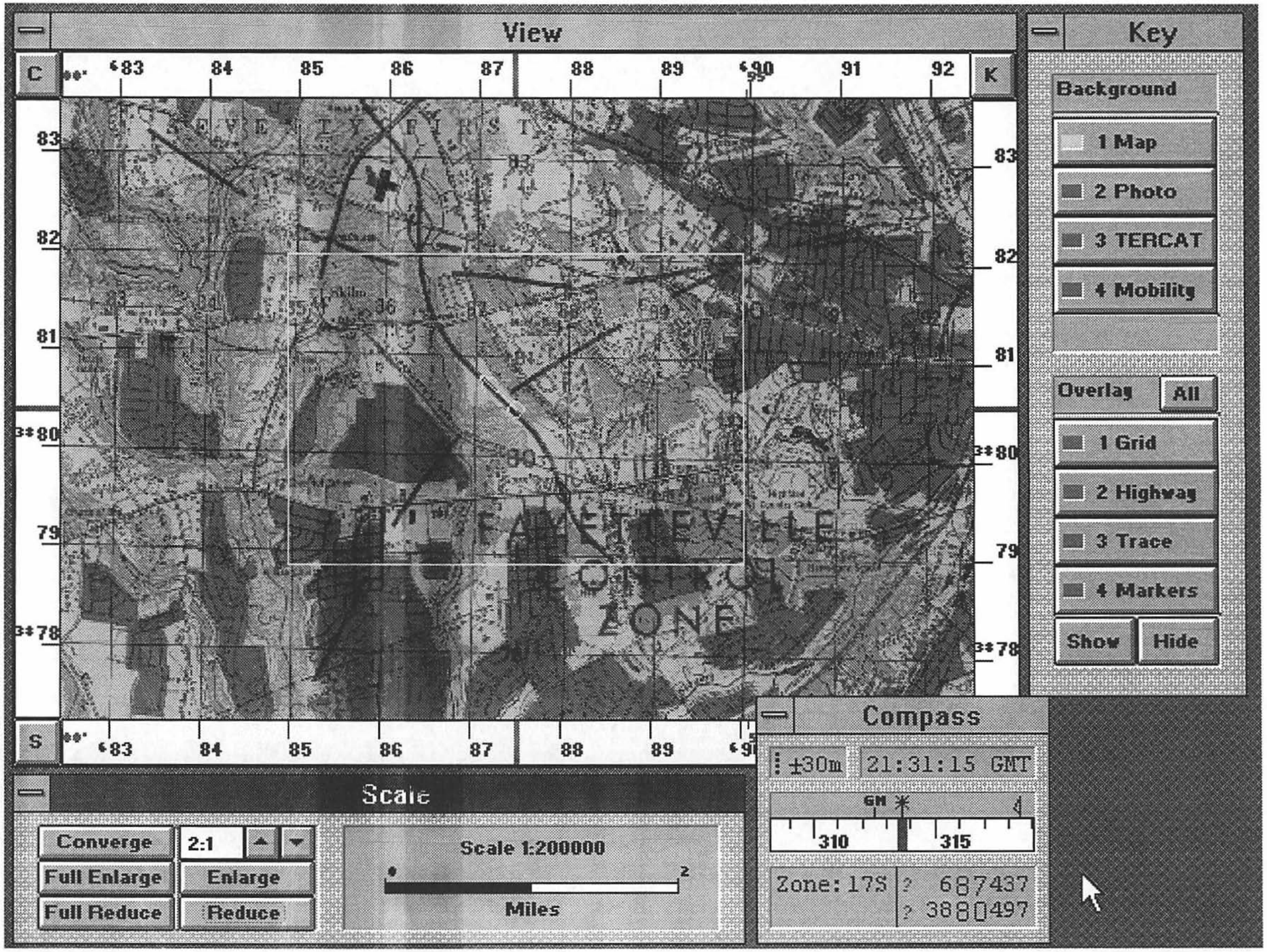
Inquiries regarding SANSE may be directed to Jeremy Salinger (`jsalinger@erim.org`) at ERIM, P. O. Box 134001, Ann Arbor, MI 48113–4001.

7. References

[1] Hilfinger, P., *Abstraction Mechanisms and Language Design*, The MIT Press, 1983.

[2] Linton, Mark A., et. al., "InterViews: A C++ Graphical Interface Toolkit," *Proceedings of the USENIX C++ Workshop*, November, 1987, Santa Fe, NM.

[3] Dennehy, T. G., "Class Libraries as an Alternative to Language Extensions for Distributed Programming," *USENIX Symposium on Experiences with Distributed and Multiprocessor Systems III (SEDMS III)*, March 26–27, 1992, Newport Beach, CA.



Scale: 2:1

Scale 1:200000

±30m 21:31:15 GMT

Zone: 17S 687437 3880497

PLATE 1. A SANSE-BASED NAVIGATION SYSTEM HAS REDUNDANT USER INTERFACES AND A COMMAND VOCABULARY OF LESS THAN 70 WORDS .

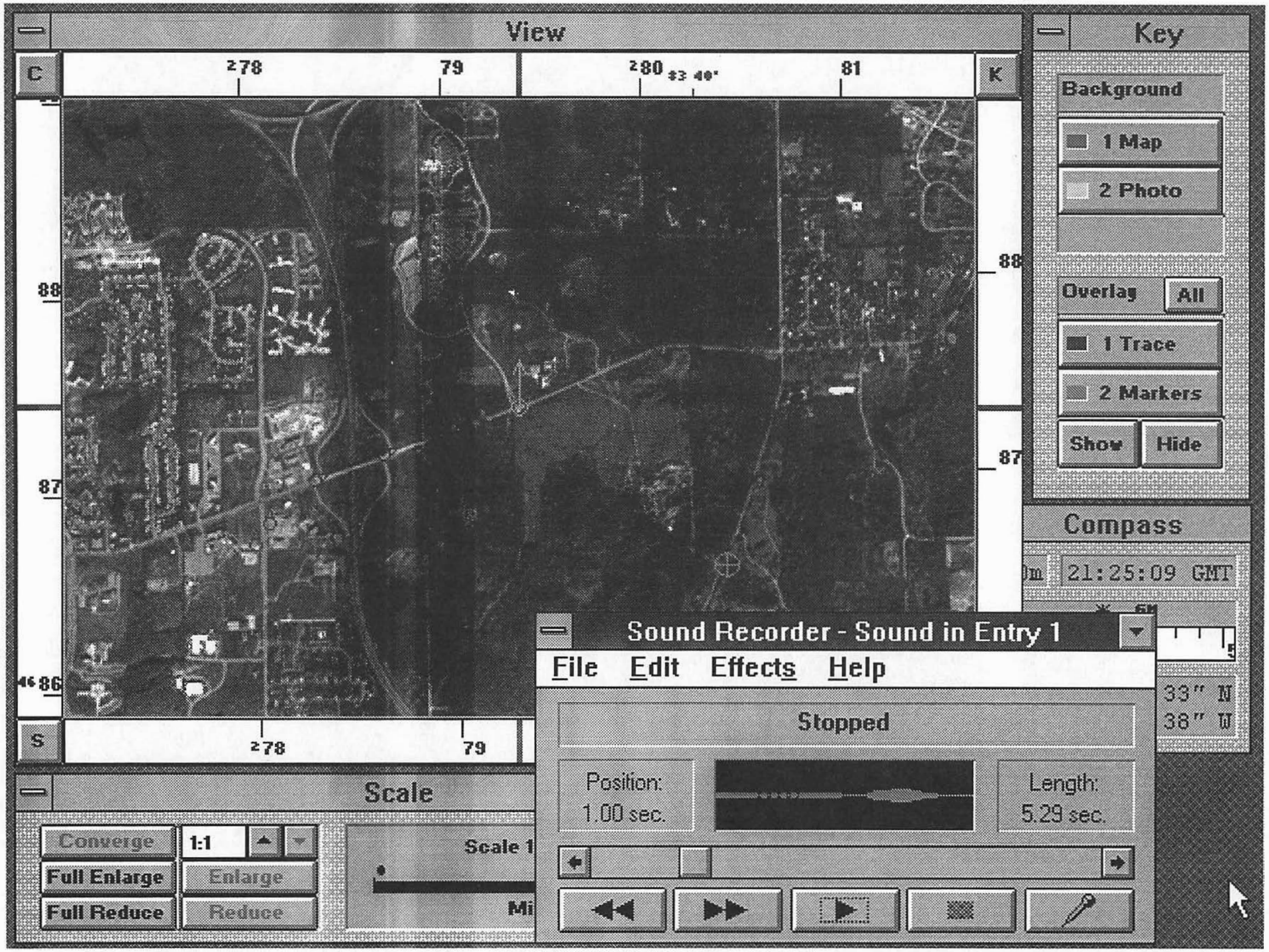


PLATE 2. SANSE USER MARKERS CAN EXCHANGE INFORMATION SUCH AS SPREDSHEET DATA AND AUDIO WITH OTHER PROGRAMS.

Energy Efficient Data Filtering and Communication in Mobile Wireless Computing

Tomasz Imielinski, Monish Gupta, Sarma Peyyeti
Dept. of Computer Science
Rutgers University
New Brunswick, NJ 08903
{imielins, monish, peyyeti}@cs.rutgers.edu

Abstract

Battery power is a scarce resource in mobile wireless computers. We discuss energy-efficient software solutions for wireless information services and general data communication. Our approach is based on allowing the receiver to be shut off when no communication is taking place and letting the CPU be in an energy efficient *doze mode* as long as possible. Minimizing transmissions (which are much more energy consuming than receiving) from mobile hosts also leads to efficient use of battery power.

We present two protocols based on this approach. The first protocol uses multicast addresses to filter out unwanted packets without waking up the CPU. Reusability of local addresses, similar to frequency reuse, facilitates using a limited multicast address space. The second protocol advocates a wake-up based scheme to save receiver power in a mobile host. We discuss the tradeoff between energy savings and increase in delay.

Experiments with implementation of these two protocols show that a significant improvement in battery usage can be achieved, making a case for certain features to be provided in hardware.

1 Introduction

Energy is a major bottleneck for mobile computers. With the very slow expected growth rate of battery life (only 20% in the next 10 years [12]) energy efficiency is a necessary feature, both on hardware and software levels. Hardware providers are beginning to offer energy efficient solutions by allowing the terminal to switch off the background light of the screen, slowing down the rotation speed of the disk (or eliminating it completely in favor of the flash cards) and offering CPUs with energy efficient *doze mode*. For example, the Hobbit processor from AT&T consumes 5,000 times less energy while in the doze mode than in the active mode (250 milliwatts in the active mode as opposed to 50 microwatts in the doze mode).

There is a growing pressure on software vendors to offer energy efficient software solutions. In this paper we provide energy efficient solutions to data filtering and subsequently generalize it to general data communication. We will assume that the mobile client is equipped with an Ethernet card (or a similar network interface device), its CPU has an energy efficient doze mode and finally, that the mobile client has the ability to switch off the receiver (this will be necessary only for the second protocol covered in section 3). We believe that such a feature, if not yet provided, will have to be offered for mobile computers[1]. In fact this paper can be treated as an argument to convince hardware providers to offer such features. The ability to switch off the receiver extended the battery life of cordless phones from a few hours to a week [4] and is also used in some pagers, which shut their receivers off periodically. A cordless phone listens to a signal from its base station only at predefined time slots and *goes to sleep* otherwise. The delay in receiving a call is not noticeable for the user at all.

The WaveLAN/PCMCIA card (a network adapter card providing wireless communication in a PC based LAN) draws approximately 3.4 Watts for transmitting and 1.8 Watts for receiving. We will show how switching off the receiver along with keeping the CPU in a doze mode could lead to dramatic energy savings.

We will consider the following two scenarios:

[1]Currently, the receiver off feature is not usually provided

- Data Filtering

  The mobile user is often a member of various multicast groups and occasionally receives large pieces of data such as news, stock quotes etc. The user is usually interested only in a small fraction of these news. For example, one may only be interested in an IBM quote, out of the whole multicast of a set of quotes for, say, 1000 companies. Since each packet of the stock multicast has the same multicast address, the software has to look at each packet of such multicast in order to filter out the IBM packet. Thus, CPU has to be active throughout the duration of the multicast. This is a waste of energy.

  We provide a method in which the MSS (Mobile Support Station) provides a much finer granularity of indexing which enables the clients to selectively listen only to a small fraction of the news. This allows the CPU to remain in the doze mode most of the time and leads to significant energy savings.

- Data Communication

  This is a more general paradigm. Currently, the client's receiver has to be *on* even if there are no messages directed to that client. We would like to be able to allow the receiver to be shut off when no messages are expected to be received by that client. This has to be fully coordinated with the MSS, analogously to the way the cordless phone synchronizes itself with its base station. Each message sent by the MSS to a client has to be first signalled by sending a wake up message during the predefined slot. Further in the paper we show that this leads to significant energy savings.

We will discuss the two concepts in the next two sections. In general the following basic modes involving the receiver and the CPU will be considered :

- Level 0: Both the receiver and the CPU are *on*
- Level 1: The receiver is *on* and the CPU is in the doze mode
- Level 2: The receiver is *off* but the CPU is *on*
- Level 3: The receiver is *off* and the CPU is in the doze mode

Higher levels reflect higher energy savings. We assume here that the CPU in the active mode draws lesser energy than the receiver. If this assumption does not hold then Level 1 is more energy efficient than Level 2. In any event, Level 3 is the most energy efficient.

We proceed first with the description of the Level 1 solution which is used for energy efficient data filtering. The Level 3 proposal is described next.

2 Receiver On and CPU in a Doze Mode: Level 1

Here we describe how we can accomplish energy efficient data filtering using Level 1 features. As in the model of Fig. 1, Server is a host on the fixed network, MSS is a fixed host with a wireless interface and MH is a mobile host communicating with the MSS over a wireless channel.

2.1 Information Dissemination in a Wireless Environment

We consider several possible strategies for information dissemination over a wireless cell. In all of the cases it is assumed that the Server to MSS communication is through periodic multicasting of data at a higher granularity. For instance, all Stock data is periodically multicasted to a *single* group address over the fixed network. This is a reasonable assumption for any scalable solution for disseminating information over a wide area internetwork.

We envision two possible different multicasting services for information dissemination. The same data can be sent using *Regular* service and the *Refined* service. Regular service provides no filtering at the MSS which simply forwards each multicast over its wireless cell. In the refined service the client is offered the ability of lower granularity filtering as well as secondary query processing, which is discussed later in this section.

1. Periodic multicasting from MSS to mobile hosts with Application Level Filtering at the mobile. Here, the MSS simply forwards the multicast it receives from the server.
2. On demand service. The mobile host transmits its query uplink and MSS replies using the data last multicast by the server. Both, the query and result are unicast (point to point) communications.
3. Mobile host transmits uplink only changes in its query status. Thus, the MSS stores the queries

for each mobile host in its cell. On receiving a new multicast from the server the MSS forwards the query results for each mobile host over the wireless cell (results can be either unicasted or multicasted).

4. Periodic multicasting from MSS to mobile hosts with Hardware Level Filtering at the mobile. This is discussed in the next subsection.

2.2 Hardware Level Filtering

This approach is based on the publisher-subscriber model of [11] with the critical difference that filtering is done at the network interface level rather than in software, enabling the CPU to stay in doze mode for longer periods of time. The server is responsible for periodically multicasting information (which can be newsgroups, stock values, etc.) over the network.

Assuming that the server gives only one general multicast address for a large collection of packets it requires that the MSS "repackage" or encapsulate the incoming multicast stream by giving smaller pieces of information separate multicast addresses.

We assume that the multicast address space will consist of permanent and transient addresses. Permanent addresses are network wide addresses allocated statically, like the network addresses of hosts on the Internet. Transient addresses are assigned from a pool of unallocated addresses. An MSS providing a refined service would choose a set of addresses from this transient address space. The client, upon joining such a refined service, will receive a form listing predefined multicast addresses as well as a hashing function for hashing queries to multicast addresses.

This hashing function is used at the MSS to map different packets to different multicast addresses depending on a primary key. The same hashing function is used at the mobile host to map primary queries to multicast addresses. The local addresses can be reused across different cells, perhaps for different services. This reusability is helpful when we have a limited number of multicast addresses.

The discussion so far applies to any information dissemination service. In the next subsection we describe an implementation of one such service, that of multicasting stock quotes. This implementation is based on hardware level filtering. Although we describe the implementation details for the stock example, the model is general and applies to any such service. Later we discuss how this model would apply to publishing newspaper columns.

2.3 Example Application - Broadcasting a Stock Database

A Stock Server running on a host in the Fixed Network periodically multicasts the stock database on a single multicast stock group address. All the hosts functioning as an MSS (base station) join the Stock Group if they are to provide the stock service to mobile hosts in their cells. Mobile hosts run client programs that provide a graphical interface for the user to query the stock database and to view the results of these queries. Fig. 1 shows the overall model for this application.

Below we describe the operation of the server, MSS and the mobile client. User queries are based on the stock's ticker tape symbol, and various descriptors and their combinations. Simplest queries involve just recent quotes of a particular stock. These are handled by using a hashing address corresponding to the name of the stock. Such queries are called ***primary queries*** because they deal with primary keys. More complex queries deal with *secondary* descriptors of stocks dealing with other attributes (secondary keys) (such as P/E ratio etc) as well as with new events such as stock value reaching a new high etc. Essentially, each query is a conjunction of "$\langle$*Attribute*$\rangle$ = $\langle$*Value*$\rangle$" conditions, where "$\langle$*Attribute*$\rangle$" can be the stock ticker symbol (a primary key) or any other secondary key (such as stock's P/E ratio etc).

Server Periodically sends all the stock packets followed by $\langle$EndOfBroadcast$\rangle$ to a well known multicast address G (Stock Group)

The **MSS** performs the following actions repeatedly

- Listen for multicasts on the Stock Group.
- Take each packet of the multicast and change its multicasting address applying the hashing function h to the stock quote contained in the packet. Store all the packets until an End of Broadcast is received.
- Form the 'Index Packets' (described later) for each descriptor using the stored packets and send them on well known Multicast Addresses over the wireless cell.
- After some delay to ensure that mobile hosts have joined the required groups, send the stored

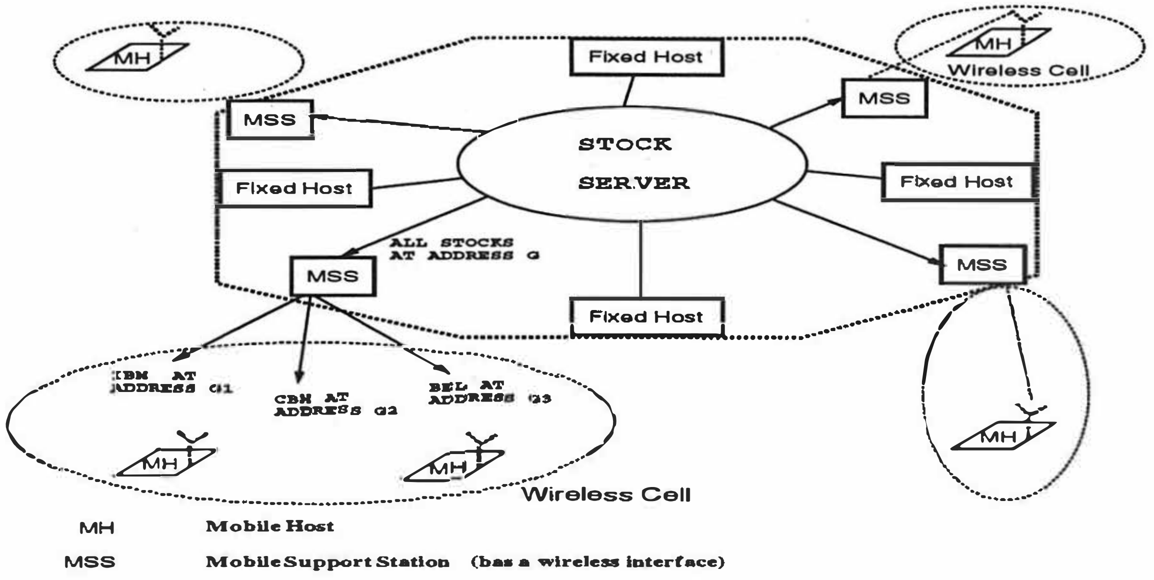


Figure 1: Model for broadcasting stock information

packets to their respective addresses. A delay of 1 second was enough in our wireless LAN environment.

- Send ⟨EndOfBroadcast⟩ on a well known multicast address.

The **Mobile-Host** performs the following actions

- Join multicast group $h(K)$ where K is a stock of interest to the user. Also join the groups for all the Index Packets that are required for the secondary queries.
- For each multicast from MSS, first get all the index packets that are required. Join the relevant addresses in the index packets.
- Read all the stock packets that arrive for the groups joined until an ⟨EndOfBroadcast⟩ is received on a well known group address.

Due to possible hashing function collisions it is still possible that the client will get a "false alarm" and the CPU will have to be woken up to conclude that after all the packet was irrelevant.

2.3.1 Hashing Function

In our implementation of the stock application, the first three characters of the ticker tape symbol are hashed into the address space of Multicast IP addresses. Multicast IP addresses are 32-bit long addresses beginning with the bit sequence 1110. Thus, a total of 2^{28} Multicast IP addresses are possible. We assumed a restricted space of the first 10 bits for the refined stock service and hash the first 3 characters of the ticker tape symbol into these 10 bits using mid-square hashing.

2.3.2 Secondary Queries

Queries based on Events and Descriptors (such as Stock reaching New High, a New Stock, etc.) are more difficult to handle. Answer to such a query returns a set of records instead of a single record. Since these descriptors are just boolean flags in the stock objects, a simple hashing scheme cannot be used, as in the case of primary queries.

Our solution is to precede the multicast of the entire stock ticker tape with a set of index packets. One index packet is sent for each descriptor. This packet contains a list of pointers - multicast addresses on which stock objects satisfying that descriptor can be received. For instance, an index node for "New High" contains a list of multicast addresses for stocks which reached a new High value.

On the client side, a query like "All objects satisfying New High " corresponds to joining all addresses in the index packet for New High. ANDing and ORing of descriptors can be done by taking intersection and union of the corresponding set of addresses. Also queries like "IBM objects with New Low" correspond to joining the address X where $< IBM, X >$ is a tuple in the New Low index packet. The index packets are sent on well known multicast addresses. A mobile host joins the groups for all the index nodes that it needs to process its queries. Fig. 2 shows an example of index packets. With this set of indexes, a mobile host interested in all Apple stocks would join the multicast address 225.6.250.1, someone interested in all stocks that reached a New High would join 225.7.1.200, 225.15.4.120 and 225.7.2.10. The query "Stocks that reached a New Low and were Split" corresponds to 225.10.0.1. The query "Microsoft stock if it reaches a New Low" will not cause the mobile host to join any groups.

2.3.3 Configuration

We have implemented the above service using Multicast-IP [3] over Ethernet, with a Wavelan network providing the wireless cell. Local addresses are used by sending IP datagrams over the wireless with a TTL value of 1. The Server runs on a SunSparcstation running SunOS, mobile hosts are laptop computers communicating with MSS using PCMCIA card. MSS is an Intel 80486 based PC-AT running Mach with a Unix server. MSS has a wavelan wireless interface and a wired Ethernet interface.

Currently the service is implemented over a single wireless cell. To incorporate host mobility between cells a *greeting* protocol needs to be implemented for the MSS to inform the mobile host about the address space and hash function for the stock service in its cell.

2.4 Another Example Application - Newspaper Publishing

Newspaper publishing is another example service which can use our protocol for energy savings. The essential difference with stock application is that the query result size is usually bigger (of the order of Kbytes). Example queries are "Getting a business column of NewYork Times", "Getting headlines of today's news" etc. Since information here is organized hierarchically, the user can issue queries based upon the results of previous queries while this is not the case in stock application.For example, the user can selectively request "hot" news items from headlines of the news.

2.5 How much energy can we save

As we said before, the energy saving is due to the client's ability to stay in a doze mode and wake up only when information which is directly relevant is multicasted. Also, due to periodic multicasting, no transmissions are needed at the mobile hosts. Transmitting being much more energy consuming than receiving, this is another reason this protocol will be energy efficient compared to conventional, demand based information delivery.

Let P_active, P_doze be the CPU power specifications for the CPU in active mode and doze mode respectively.

Let R_active, $R\_mismatch$ be the power consumed by the receiver for receiving a packet and for filtering out a packet (due to address mismatch) respectively. T_active is the power consumed in transmitting.

Usually, $P\_active \gg P\_doze$ and $T\_active > R\_active$.
We assume that $R\_active \gg R\_mismatch$, i.e. power required to detect that a packet is not destined for us is relatively negligible, as only the address portion of the packet needs to be read.

Further, let
N be the size of database being multicasted in number of packets,
BW be the bandwidth over the wireless Cell (in bytes/sec),
P be the size of each packet (in bytes),
R be the average number of packets a user is delivered as query results,
Q be the average query size (in bytes) generated by each client between successive multicasts.
X be the average time CPU needs to be in the active state to transfer one byte of data between the application software and the network interface.

1. Application Level Filtering at the mobile.

   CPU Energy consumed = $(N * P * X) * P\_active$

   Receiver Energy consumed = $(N * P/BW) * R\_active$

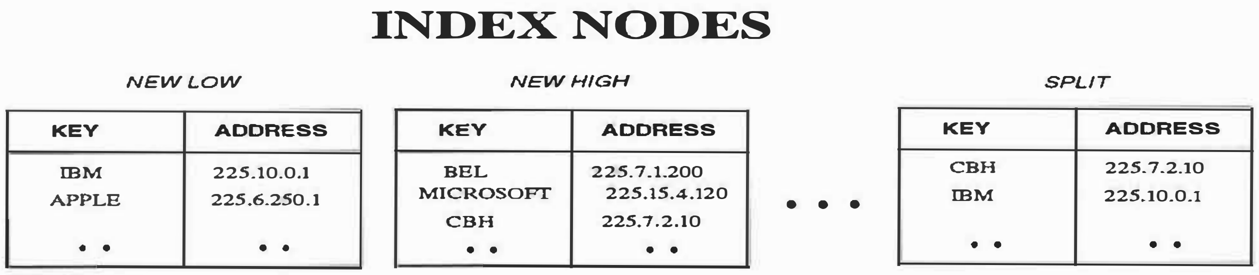
2. On demand service.

   CPU Energy consumed = $(R*P*X)*P\_active+ (Q * X) * P\_active$

   Receiver Energy consumed = $(R * P/BW) * R\_active$

   Transmitter Energy consumed = $(Q/BW) * T\_active$

3. Storing queries at MSS.



NEW LOW

KEY	ADDRESS
IBM	225.10.0.1
APPLE	225.6.250.1
• •	• •

NEW HIGH

KEY	ADDRESS
BEL	225.7.1.200
MICROSOFT	225.15.4.120
CBH	225.7.2.10
• •	• •

SPLIT

KEY	ADDRESS
CBH	225.7.2.10
IBM	225.10.0.1
• •	• •

Figure 2: Index Packets

Let F be the average fractional change in queries at each client in time T. So, $F * Q$ bytes are transmitted by a mobile host to inform the MSS of its change in queries.

CPU Energy consumed = $(R*P*X)*P\_active+(F*Q*X)*P\_active$

Receiver Energy consumed = $(R*P/BW)*R\_active$

Transmitter Energy consumed = $(F*Q/BW)*T\_active$

4. Hardware Level Filtering at the mobile.

   Let L be the average number of overhead packets. These are the index packets received and packets received due to collisions. These collisions could be due to the hashing function used in the application or due to collisions occurring in translating multicast addresses (Link Level and IP Level Collisions [3]).

   CPU Energy consumed = $((R+L)*P*X)*P\_active$

   Receiver Energy consumed = $((R+L)*P/BW)*R\_active+((N-R-L)*P/BW)*R\_mismatch$

Schemes (2),(3) are demand-based while (1),(4) are based on periodic multicasting. (3) is obviously more energy efficient than (2) and (4) is more energy efficient compared to (1). Assuming $R\_mismatch \ll R\_active$ and $P\_doze \ll P\_active$, it can be derived that (4) is more energy efficient than (3) if

$$L/(F*Q) < (1/P) * \frac{P\_active*X+T\_active/BW}{P\_active*X+R\_active/BW}.$$

Thus, for larger collision rates (3) would be better than (4), whereas (4) is more energy efficient for large values of F. For applications such as newspaper browsing frequent changes in queries will be a common scenario. For large cell sizes, higher $\frac{T\_active}{R\_active}$ ratio is in favor of (4) being more energy efficient than (3).

Also, in (3) queries for all the hosts are stored at the MSS. When a mobile host changes cell, this state information will either have to be transferred to the new MSS or the mobile will need to transmit all its queries uplink to the new MSS.

2.6 Conclusion

To summarize, we have achieved the level 1 of power saving by providing a more refined data filtering to the clients on the *network level*. This allowed us to keep the CPU in a doze mode and use the higher levels of the protocol stack only in case when the packet relevant to the user's query is received. We have demonstrated how multicast addresses used for the finer granularity indexing can be reused in different cells by doing encapsulation of data at the MSS.

3 Receiver Off and CPU in the Doze Mode: Level 3

In this section we show how level 3 energy saving mode can be implemented for applications which do not require a real time response, and can tolerate some delay. E-mail and news group reading belong to this category. The ideas here are a direct generalization of the previously mentioned energy efficient features used in cordless phones.

3.1 Preview Edition Protocol

The communication channel between the MSS and a Mobile Host is assumed to have the logical layout depicted in Fig. 3

Each mobile host keeps its receiver on during the *preview* period. MSS constructs a *wake-up* packet during the preview and broadcasts it to all mobile hosts in its cell. The wake-up packet contains a list of addresses of those hosts for which messages will

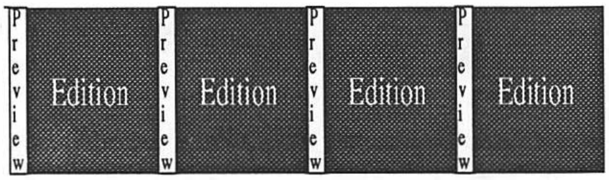


Figure 3: Preview Edition Cycles

be sent during the following *edition* period. The mobile host keeps its receiver on only if it receives the wake-up packet and finds its address in the list. Otherwise, it turns the receiver off until the next preview period.

For each message that the MSS needs to send to a host, it checks if the wake-up packet for the current preview-edition cycle has been sent and if it contained the host's address. If so, the message is forwarded immediately, else it is buffered until the next wake-up packet has to be sent. This necessitates any receivers that were left *on* to stay *on* for the entire edition period.

In case there are messages to be multicasted to a group, the MSS adds the corresponding multicast addresses in the wake-up packet. The mobile host not only checks for its own address but also the multicast addresses for the groups to which it belongs.

Let *layout ratio* r be defined as $r =$

(size of preview period)/(size of edition period)

and *activity ratio* A be defined as $A =$

fraction of time the receiver is *on*

Clearly, lower the activity ratio, higher is the energy saving obtained from the preview-edition protocol. If the mobile host receives no messages, it can keep the receiver off for $1/(1+r)$ fraction of total time. So, $A \leq 1/(1+r)$. This leads to very significant energy savings as compared to the current situation when such a host would have to keep the receiver on, leading to unnecessary waste of energy.

Intuitively, longer edition time leads to longer average delay of message delivery. For instance, if the edition time is 9 minutes, the message received by the MSS, while it has already started dispatching messages in the current edition, can be delayed 9 minutes and possibly more. The same layout ratio r can be implemented in many different ways ranging from very short preview and edition sizes to much longer previews and editions. For example, the layout ratio $r = 0.1$ could be implemented by the preview of 1 second and the edition time of 10 seconds, as well as by the preview time of 1 minute and the edition time of 10 minutes. The latter solution leads to much longer delays per message but has an advantage of the longer preview time. Such longer preview time minimizes the probability of the client losing the wake up call[2].

Below we describe the experimental results in which we varied the layout ratio, message volume and the sizes of the preview, edition periods. We have measured the average delay and the activity ratio.

3.2 Experiment

The experiments have been performed on an Ethernet based LAN. The base station and the client (mobile host) are simulated as two processes on different machines in the LAN. The base station generates messages at a given rate. In each preview period, if there are any messages pending in its message queue, the base station sends a wake-up call to the client. The wake-up call is implemented using a tcp-connection. If the base station is unable to connect to the client during this period, the wake-up call is considered missing. This might happen due to other traffic in the LAN. In each edition, if wake-up call is sent, base station sends messages in its queue using a tcp-connection. Once a wake-up call is sent, messages generated within an edition do not have to wait till the next preview. The client waits for wake-up calls during each preview period. If there are any, it waits for messages during the following edition, otherwise it sleeps.

For the purpose of measuring the delay, birth-time stamps are attached to each message when it is generated. When the client receives the message it sends a reply to the base station which then measures the delay incurred in delivering the message. Overhead for shutting the receiver off and on is taken into account in measuring the fraction of time the receiver is off. For a given layout ratio and message rate, preview and edition are varied to measure the average delay and the activity ratio. Measurements are taken for High(1:10) and Low(1:100) layout ratios and High(120/hour) and Low(3/hr) message rates for messages of size 2Kbytes. Receiver shutoff overhead is the time it takes to configure the interface unit for reception when the unit is powered on. It is assumed to be of the order of a few milli-seconds.

[2]In Ethernet we can not guarantee the exact time of the message delivery, hence a very short preview time could result in a significant message loss

layout ratio 1:10 message rate 120/hr			
preview secs	edition secs	avg delay secs	activity ratio
0.5	5	6.080	0.258
1	10	5.760	0.360
5	50	18.654	0.483
10	100	34.937	0.545

layout ratio 1:100 message rate 120/hr			
preview secs	edition secs	avg delay secs	activity ratio
0.05	5	5.145	0.175
0.1	10	10.68	0.316
0.5	50	28.621	0.547
1	100	30.083	0.519

layout ratio 1:10 message rate 3/hr			
preview secs	edition secs	avg delay secs	activity ratio
0.1	1	1.107	0.092
1	10	15.950	0.099
5	50	18.215	0.133
10	100	35.666	0.176

layout ratio 1:100 message rate 3/hr			
preview secs	edition secs	avg delay secs	activity ratio
0.1	10	12.253	0.018
0.5	50	59.014	0.052
1	100	129.274	0.094

3.2.1 Interpretation of Results

Low message rates[3] result in receiver being off for upto 98% of the time. Hence, a message generated will have to wait till the next preview thus increasing average delay. Within the same layout ratio, for low preview sizes, the delay is comparable to size of edition since probability of missing the wake-up call is high. Lower layout ratios introduce larger delays because probability of no messages generated during preview is higher.

For high message rates, average delay did not grow proportional to the edition size.This is because once wake-up call is sent for an edition, messages generated during that edition do not have to wait till the next preview as the receiver is on in this edition. The activity ratio is high compared to low message rates. Layout ratios have lesser impact on the delay.

3.2.2 Conclusion

We find that implementing the level 3 of energy savings can, in practice, lead to quite dramatic energy savings. The Wavelan's receiver draws 1.8 W when on and 0.18 W when it is in doze mode (the card still has to be powered even if the receiver is off). The average e-mail and newsgroup user receives probably at most 30 to 40 messages a day and probably does not mind a few minute delay.

For instance, with the layout ratio $r = 0.01$, implemented with a preview size of 100 milli-seconds and an edition size of 10 seconds with message delay of 12 seconds, we could reduce the total energy consumption due to the activity of the receiver by an order of magnitude. For example, assuming that the mobile terminal is powered by 4 AA batteries each of which is capable of providing around 1 Watt for an hour we can keep the Wavelan receiver on for about 2.25 hours (and this ignores all the power drawn by other hardware elements!). If the ratio $r = 0.01$ leads to the activity of 2% then the same terminal can be receiving messages for about 19 hours.

We have ignored here the possible energy saving due to the CPU being in the doze mode. These savings are application dependent, since the CPU may have to run other applications which are not communication dependent. However, if the client is "communication intensive" then the savings due to the CPU being in the doze mode will have to be included as well.

[3]3 e-mail messages per hour constitute a low message volume from the hardware point of view, but actually a fairly intense e-mail use from the user's perspective.

3.3 Implementation of The Preview-Edition Protocol

In this subsection we discuss issues regarding which layers of the TCP/IP protocol stack should be made aware of the preview edition protocol. We take E-mail as an example application. It should be ensured that no changes are required in the existing network software on the Fixed Hosts in the Internet (except for the Mobile Support Stations).

Implementing the preview-edition protocol at the Application Layer means there are System Calls available : shutoff() and wakeup() for switching the receiver off and on at the Mobile Host. At the MSS, application level software is needed to buffer messages (say, e-mail messages) until the next preview period, send destination IP Addresses during the preview and then send the messages during the edition period.

This obviously works only if there is an Indirect Protocol running at the Application layer. For example, for E-mail, currently a TCP connection is established between sender and receiver hosts. Now, there will have to be a RemoteHost-MSS-Mobile E-mail protocol. The MSS will receive E-mails on behalf of the Mobile Host and forward them using preview-edition.

On the other hand, this protocol can be implemented at the link layer. For this, any packets received by the link layer at MSS (say, Ethernet) will be buffered until the next preview period, their destination link-layer addresses will be sent during preview and the buffered packets are sent during edition. How this affects the throughput needs to be studied.

3.4 Applications of the Preview-Edition Protocol

Here we would like to discuss some of the possible applications of the preview-edition protocol. Our main motivation is, in fact, the electronic mail, but there are a number of related applications such as *talk* and *news*.

The main advantages of our protocol deal with the situations when communication is initiated by the MSS, not by the mobile host. Thus, applications like telnet and ftp, which are initiated by the client, are not going to benefit from our solution. In these cases we assume that the mobile host will stay active and will keep the receiver on during the (ftp, rlogin, telnet) session. On the other hand, in case of e-mail, we see dramatic savings. Instead of keeping the host's receiver on all the time, the receiver will be turned on and off periodically to listen to the preview report. If no messages are waiting at the MSS, then the receiver will be off for a large fraction of time (depending on the parameters of the protocol but roughly 90% or more). On the other hand, if the preview contains a *biff* message then the mobile host will have two basic choices: to request explicitly that the mail is downloaded from the MSS or to wait until the preview-edition protocol takes care of it without an explicit uplink request. The first solution corresponds to the mobile host executing "mail" command and the mail being downloaded by the MSS only upon receiving such a request from the host. We argue, in general, against such a solution: it will unnecessarily increase the uplink message traffic on the narrow uplink channel. In our solution, the MSS will anyhow do the best effort downloading of the message to the mobile host as soon as it can. In fact, the MSS should not even wake up the mobile host if it is not able to download the message in the next edition. In such a case the wake up message will be delayed until the MSS is capable of fulfilling the promise. Thus, there is no point in rushing the MSS by sending the mobile host initiated message, since the MSS cannot do better anyway. Notice that the user still has freedom to read his mail at any time. It is just that execution of the mail command will not result in the uplink message but rather in the local mail buffer access. The explicit uplink request is useful only in situations when the user wants to delay reading his messages until he is connected again. In such a case, waiting for the user's request can save wireless bandwidth. This assumes, however, that the message is not considered to be "urgent". We assume that the decision which messages are urgent and which are not has been made by the mobile user prior to "going wireless" and that the MSS is only considering urgent messages for wireless transmission anyhow. Yet another solution is that MSS include only message headers in the edition period and then allow the mobile user to explicitly request the selected subset of messages. In this case, the headers are received without the uplink request, while the messages are delivered to the mobile host responding to the explicit uplink request.

For applications like "talk" which are more real time, the client should probably switch on his receiver for the entire duration of the talk session. Otherwise, the energy saving is not enough to justify the increase in delay. Frequently switching the receiver on and off is an added overhead contributing more delay.

News applications (newsgroups, newsgroup updates) are more similar to e-mail and can be treated in a similar way. Perhaps the user can specify "urgent and important" news which have to be delivered as soon as they arrive and second priority news which can be wait to be seen after the client connects again.

3.5 Related Work : Differences and Similarities

As we said earlier, the preview-edition protocol has been motivated by the similar protocol used in cordless telephony. In satellite based paging solutions Motorola and Olivetti are considering keeping the pager's receiver "on" only at predefined periods of time which depend on the pager's identification. Therefore the activity periods are different for different classes of pagers. Also, there is no concept of edition and the solution is considered only for satellite based networks. In our case, we would like to implement it on variety of platforms, including wireless Ethernet and possibly, in the future, CDPD.

A similar protocol is mentioned in [7]. There, the mobile host keeps its receiver off until a message is to be received from the base station. At this time an uplink request is sent to the base station, indicating that the mobile unit is ready to receive data. This scheme would not work if the base station needs to send messages to the mobile unit asynchronously, i.e. without the mobile expecting them. A wake-up based protocol is required in such a case.

Our protocol has a "reservation protocol" flavor, but the concept of the reservation is very loose. Indeed, the reservation is only a "promise" of the MSS to deliver the packet (sometime) within the next edition. This promise could be broken. The only "reserved" slot is the preview period and this is the slot which is "polled" by the mobile clients.

Notice, the asymmetric nature of the preview-edition protocol which covers only the downlink channel. The uplink communication is regulated separately, since we are not concerned with the MSS power consumption. However, the preview-edition protocol reduces the uplink network traffic, if we avoid the explicit uplink requests by the mobile hosts (as recommended).

4 Conclusions

We have proposed two protocols which offer significant energy savings for applications such as newsgroups and electronic mail. The first protocol provides refined indexing of multcasted data and allows energy savings due to keeping the CPU in the doze mode. The preview-edition protocol allows, additionally, to keep the client's receiver off for most of the time. We have provided experimental results by implementing the above protocols on the application layer and testing them in a wireless LAN setting.

One can visualize a very simple combination of both protocols, when the client has to be first woken up before receiving the multicast group news. Further filtering within the multicast itself can be accomplished using the hashing based protocol described before.

We have implemented both protocols on the application layer, assuming that the features of switching the receiver off and forcing CPU into the doze mode are provided by the operating system.

References

[1] Ajay Bakre, B.R. Badrinath, "I-TCP: Indirect TCP for Mobile Hosts", Technical Report DCS-TR-314, Department of Computer Science, Rutgers University, October 1994.

[2] T. F. Bowen et al., "The DATACYCLE Architecture," *Communications of the ACM*, Vol. 35, No. 12, pp. 71-81, December 1992.

[3] Stephen Deering, "Multicast Routing in a Datagram Internetwork," *phd thesis*, Stanford University, December 1991.

[4] R. Frenkiel "Private Communication"

[5] David Gifford et. al., "The application of digital broadcast communication to large scale information systems," *IEEE Journal on selected areas in communications*, Vol 3, No. 3, pp. 457-467, May 1985.

[6] David J. Goodman, "Trends in Cellular and Cordless Communications," *IEEE Communications Magazine*, pp. 31-40, June 1991.

[7] Andy Harter and Frazer Bennett, "Low Bandwidth Infra-Red Networks and Protocols for Mobile Communicating Devices", Technical Report tr.93.5, Olivetti Research Laboratory.

[8] T. Imielinski and B. R. Badrinath, "Mobile Wireless Computing: Challenges in Data Management," *Communications of the ACM*, Vol. 37, No. 10, October 1994.

[9] T. Imielinski, S. Viswanathan and B. R. Badrinath, "Energy Efficient Indexing on Air," *Proc of ACM-SIGMOD, International Conference on Data Management*, Minnesota, pp. 25-36, May 1994.

[10] T. Imielinski, S. Viswanathan and B. R. Badrinath, "Power Efficient Filtering of Data on Air," *Proc of 4th International Conference on Extending Database Technology*, Cambridge-U.K, pp. 245-258, March 1994.

[11] B. Oki et al., “The Information Bus - An architecture for Extensible Distributed Systems,” *Proc of ACM-SIGOPS*, December 1993.

[12] Samuel Sheng, Ananth Chandrasekaran, and R. W. Broderson, “A portable multimedia terminal for personal communications,” *IEEE Communications Magazine*, pp. 64-75, December 1992.

[13] Lee, William C. Y., “Mobile Cellular Telecommunications Systems,” *New York: McGraw-Hill*, 1989.

[14] J.W. Wong., “Broadcast Delivery,” *Proceedings of the IEEE*, Vol. 76, No. 12, pp. 1566-1577, December 1988.

Adaptive Disk Spin-down Policies for Mobile Computers[*]

Fred Douglis[†]
AT&T Bell Laboratories
P. Krishnan
Brown University
Brian Bershad
University of Washington

Abstract

Mobile computers typically spin down their hard disk after a fixed period of inactivity. If this threshold is too long, the disk wastes energy; if it is too short, the delay due to spinning the disk up again frustrates the user. Usage patterns change over time, so a single fixed threshold may not be appropriate at all times. Also, different users may have varying priorities with respect to trading off energy conservation against performance. We describe a method for varying the spin-down threshold dynamically by adapting to the user's access patterns and priorities. *Adaptive spin-down* can in some circumstances reduce by up to 50% the number of disk spin-ups that are deemed by the user to be inconvenient, while only moderately increasing energy consumption.

1 Introduction

In today's mobile computers, the hard disk is typically spun down after a fixed period of inactivity in order to conserve energy. When the disk is next accessed, it is spun up again, which can cause a delay of a few seconds. This spin-up delay may be acceptable to the user, who knows that the delay is in exchange for extending battery life, or it may be bothersome. We have developed a method for distinguishing between *undesirable* and *acceptable* spin-up delays and varying the idle-time threshold for spinning down the disk based on the user's tolerance for undesirable delays. We term this method *adaptive disk spin-down*, and henceforth refer to undesirable delays as "bumps" for simplicity.

A good adaptive policy will reduce the number of bumps without adversely affecting energy consumption compared to a fixed-threshold policy; or it will reduce energy consumption without adversely affecting the number of bumps. In the best case, it will improve both of these metrics; however, as explained in Section 5.6, that is difficult to achieve.

The rest of this paper is organized as follows. The next section discusses disk spin-down in greater detail. Section 3 describes adaptive spin-down and defines some terminology. In Section 4, we describe the experiments we performed, and in Section 5 we report their results. Section 6 summarizes related work, and Section 7 concludes.

2 Background

2.1 Disk Spin-Down

The functionality of mobile computers is limited by the amount of time they can operate on a single battery charge. Most mobile computers use magnetic disk drives, which can consume 20–30% of total system power, or more [5]. The power consumed by the disk subsystem can be reduced by spinning the disk only when necessary. All systems of which we are aware use a *fixed-threshold policy* to spin down the disk: if the disk has not been accessed in T seconds it is spun down. The disk is spun up again the next time it is accessed, which delays the access by 1–2s or more while the disk readies itself.

In general, a spin-down policy has two conflicting goals: reducing energy consumption and preserving interactive response. Spinning down the disk after a short period of inactivity can decrease energy consumption but will also result in more delays due to spin-up. Thus the fixed threshold T is typically on the order of many seconds or minutes to minimize the delay from on-demand disk spin-ups. Another reason for a large fixed threshold is that is possible to consume so much energy due to the spin-ups that overall energy consumption increases with a shorter threshold. After being spun down, the disk must stay spun down for a period of time in order to amortize the spin-up overhead. The break-even point, T_d, de-

[*]This work was performed in part at the Matsushita Information Technology Laboratory of Panasonic Technologies, Inc.
[†]Email: douglis@research.att.com

pends on the time and energy consumed in each state and can be statically calculated based on the parameters for a disk [5]. A fixed threshold shorter than T_d can in some cases increase energy consumption, and the closer the threshold gets to spinning down the disk immediately, the greater the likelihood of increasing rather than decreasing energy consumption.

The Hewlett-Packard Kittyhawk C3014A spins down and up again in about 3s, and its manufacturer recommends spinning it down after about 5s of inactivity [8]; most other disks take several seconds for spin-down/spin-up and are recommended to spin down only after a period of about 5m [3, 19]. The Quantum Go•Drive in most Macintosh PowerBooks takes approximately 5s to spin up, and the PowerBook by default allows the drive to be spun down after 30s to 15m of inactivity, or not at all. Commercial products such as Connectix PowerBook Utilities [2] allow finer-grained control over the threshold, however.

The main intuition underlying the variations among disk spin-down policies is identifying periods of inactivity at disk that are "sufficiently large." Fixed-threshold policies wait a constant period of time to be sure that the period of inactivity is large enough. Compared to thresholds of 30s or more usually recommended by manufacturers, aggressive fixed-threshold spin-down policies—those that spin down the disk after a relatively short period of inactivity—often consume less energy because they observe inherent characteristics of disk interarrival times when typical activities are performed by the user [5, 14]. Usually, either the disk is accessed repeatedly in a short time-span, or it is idle longer than $T_d + \Delta$, where Δ is the time one waits before spinning down the disk. The greater the disk interarrival times, the more effective an aggressive spin-down policy can be.

2.2 Caching

A simple method that increases the interarrival times at disk and helps nearly[1] any disk spin-down policy to decrease energy consumption is to reduce the number of activities at disk via caching or buffering of I/O. Both DRAM caching and SRAM buffering affect which I/Os go through to the disk. DRAM serves as a cost-effective cache for read-only data and can dramatically reduce energy consumption and improve performance [5, 14]. It does have the potential to cause additional spin-up delays compared to a configuration with less DRAM caching, because misses in the DRAM cache are more likely to find a spun-down disk.

[1]There are situations when caching can eliminate just enough I/Os that the disk spins down due to inactivity, but not enough to amortize the cost of the spin-down. In this case, the overhead from spinning up the disk can increase over-all energy consumption.

SRAM can absorb writes to disk, which decouples disk latency from application performance when a synchronous write can be performed to SRAM rather than to disk [1]. It can reduce energy consumption by avoiding writes to disk completely if the same blocks are frequently overwritten. If we assume that SRAM is completely recoverable in case of a system crash or other failure, I/Os to SRAM need go to disk only when SRAM is full. In this case, writes to disk can sometimes be completely eliminated (for example, when data blocks are overwritten), and the disk may spin down when it otherwise would have been accessed for writing. As with DRAM caching, subsequent operations (reads or writes) may be delayed by a disk spin-up.

One feature of deferring writes to disk indefinitely is that if the disk is currently spun down, a write that fits in the SRAM buffer need not spin up the disk at all. While many disks use SRAM as a write buffer, we know of only one, the Quantum Daytona, that buffers writes to a spun-down disk rather than spinning it up again. Because aggressive spin-down policies and relatively short spin-up delays will result in the disk being spun down more often than on past systems, and deferring small writes will be extremely important, we consider a Daytona-style disk drive in this study. We show the impact of deferred writes later in this paper by comparing it to a policy that quickly writes blocks from SRAM to disk.

Although the discussion in this paper focuses on mobile computers, which have limited battery life on a single charge, desktop computers can benefit from these techniques as well. Manufacturers are striving to provide low-power desktop machines [7]; spinning down a disk on a desktop computer is one necessary aspect of the EPA Energy Star Computers Program [9]. Adding SRAM to a workstation can improve performance and reduce server load [1].

2.3 Evaluation Metrics

There are several possible metrics by which one may evaluate a spin-down policy. One simple metric is *least energy*, which optimizes energy savings with no regard to spin-up delays. At the other extreme is the policy that minimizes spin-up delays. In the absence of future knowledge of accesses, the latter policy is one that never spins down the disk, minimizing delays but usually consuming substantially more energy.

Numerous metrics lie between these two extremes. One possible method is to distinguish between spin-up delays that the user finds acceptable and those that severely inconvenience the user. Towards this end, we introduce the notion of *undesirable spin-ups*, or *bumps*, and formalize a possible definition of bumps in Section 3.1. As an example, a user who must wait for the disk to spin up when accessing the computer for the first time in 30m should find a short delay an acceptable cost of saving the energy needed to spin the disk for the entire time. By comparison, most users will be irritated if they must wait a few seconds for the disk to spin up when it has only been idle a few seconds. Compared to the energy-optimal spin-down threshold, a small increase in the threshold T can substantially decrease the number of spin-ups relative to the increase in energy consumption.

3 Adaptive Spin-down

We define *adaptive spin-down* as a policy that monitors the spin-down threshold and adjusts it to keep a balance between energy consumption and bumps. In this paper we consider a policy that attempts to keep the number of bumps within a tolerable range (it need not necessarily be zero). The user defines what is acceptable and what is undesirable. We first refine the notion of an undesirable spin-up and then describe how to adjust the spin-down threshold dynamically.

3.1 Undesirable Spin-ups

There are many different ways of defining acceptable spin-ups; in this paper, we define our measure of acceptability as a function of the ratio ρ between the spin-up delay δ and the idle time I of the disk prior to the spin-up. A spin-up delay is *unacceptable* (i.e., a bump) if $\delta > \rho I$. A value of $\rho = 0$ indicates that all spin-up delays are considered bumps.

A way to ensure that there are *no* bumps is to use a fixed spin-down threshold $T = \delta_M / \rho$, where δ_M is the maximum delay possible (including any overhead due to spinning down the disk first, if the I/O occurs just as the disk starts to spin down). In other words, one would wait until the disk has been idle long enough that spinning it up again cannot be perceived as an inconvenience. While this policy will satisfy any metric that tries to minimize the number of bumps, it may consume much more energy than a policy that allows a small number of bumps.

3.2 Threshold Adjustment

Here we describe a software approach to adaptive disk spin-down. *Threshold adjustment* is at the heart of adaptive spin-down. The adjustment may take place at various times: when a bump occurs, when an acceptable spin-up occurs, or when other information suggests the need to change the threshold.

- When a bump occurs, the threshold was too short and should be increased.
- When a spin-up is acceptable, the threshold was long enough. It can possibly be decreased without increasing the number of bumps.
- There may be other times when the threshold should be changed even though no spin-up has occurred. For instance, if the disk is idle I seconds and the current spin-down threshold T is just greater than I, i.e., $I < T < (1+\epsilon)I$, then the disk will not be spun down. But a slight variation in the time of the next disk I/O could cause the disk to be spun down, after being idle for more than T seconds, only to be spun up again immediately. Thus if there is a *close call*, in which the spin-down threshold was barely high enough to avoid a situation in which the disk would be spun down and back up again in a short time, it may be appropriate to increase the threshold.

There are many possible approaches to, and enhancements of, adaptive spin-down. They include:

Rate of adjustment By how much is the threshold adjusted when a spin-up occurs? The formula for adjusting the threshold can be arbitrarily complex. One example is to add two different values, α_a and β_a respectively,[2] to T when undesirable or acceptable spin-ups occur. One would expect that $\alpha_a > 0$, $\beta_a < 0$, and $\alpha_a > |\beta_a|$. In other words, when a bump occurs, the spin-down threshold should be increased by enough to make future bumps significantly less likely; when an acceptable spin-up takes place, the spin-down threshold should be decreased, but more gradually.

Another related method is to multiply T by α_m and β_m respectively. Here one expects that $\alpha_m > 1$ and $1 > \beta_m \geq 1/\alpha_m$. This assumption is examined more closely in Section 5.2.

Restrictions on threshold In order to avoid pathological behavior, adjustments to the spin-down threshold may apply only within certain ranges. The minimum spin-down threshold may be 0s (spin the disk down immediately upon each access), or it may be positive in an attempt to keep the adjustment process from "overdoing" its compensating behavior. The minimum value is especially important if ρ is high, since there is an implicit assumption above that the threshold can be decreased whenever an acceptable spin-up occurs. This assumption will not

[2]The subscript a denotes an additive adjustment and the subscript m denotes a multiplicative adjustment.

	MAC			DOS			HP		
Applications	Finder, Excel, Newton Toolkit			Framemaker, Powerpoint, Word			email, editing		
Duration	3.5 hours			1.5 hours			4.4 days		
Number of distinct Kbytes accessed	22000			16300			32000		
Fraction of reads	0.50			0.24			0.38		
Block size (Kbytes)	1			0.5			1		
Mean read size (blocks)	1.3			3.8			4.3		
Mean write size (blocks)	1.2			3.4			6.2		
Inter-arrival time (s)	Mean	Max	σ	Mean	Max	σ	Mean	Max	σ
	0.078	90.8	0.57	0.528	713.0	10.8	11.1	30min	112.3

Table 1: Summary of trace characteristics. The statistics apply to the 90% of each trace that is actually simulated after the warm start. Note that it is not appropriate to compare performance or energy consumption of simulations of different traces, because of the different mean transfer sizes and durations of each trace. (This table is reproduced from [4].)

hold if spinning down too quickly results not only in many spin-up delays (which the user has said are acceptable) but also an increase in over-all energy consumption.

Similarly, bumps may increase the threshold indefinitely, or they may be ignored after some point. As mentioned above, there is no point to increasing the threshold beyond δ_M / p if one is trying to minimize bumps.

Time of adjustment The method described above adjusts the threshold upon every spin-up event, as well as after "close calls." This approach can be modified, for example, to decrease the spin-down threshold only when several spin-ups in a row are acceptable.

Variable penalties Thus far, the description of adaptive spin-down has assumed that either a spin-up is acceptable, or it is not. There is in fact a continuum of degrees of acceptance: spinning up just after the disk has spun down is worse than spinning up just before the point at which the spin-up delay would be deemed acceptable. In fact, the *worst* type of spin-up is one that occurs before the disk has completely spun down, since the delay will be greater (the disk must typically be fully spun down before the order to spin up can be issued). An improvement on the method above, therefore, is to increment the spin-down threshold by a greater amount when the delay due to a spin-up is especially egregious than when it is barely above the user's threshold.

4 Experiments

The effectiveness of adaptive policies depends on a number of factors: workload, hardware characteristics (i.e., disk parameters, DRAM size, and SRAM size), user perception (i.e., the acceptability ratio), and threshold adjustment ranges and modifiers. We performed several experiments to quantify the effect of these factors. To keep the study manageable, we fixed DRAM at 1 Mbyte and SRAM at 32 Kbytes except where noted.

To study workload, we simulated both adaptive and fixed-threshold spin-down policies on three traces used in a previous study of mobile storage management: a 3.5-hour Macintosh PowerBook trace, a 1.5-hour Windows 3.1 trace, and a 4.4-day HP-UX trace [4]. Table 1, reproduced from that study, summarizes information about the traces. In addition, the MAC trace has a distinctive quality that has a significant impact on the effectiveness of the SRAM write buffer: it has a very high locality of write accesses, with 36% of writes going to just one 1-Kbyte block and 24% of writes going to another. Note as well that the HP trace contains disk-level rather than file-level accesses, below the level of the buffer cache, so we do not simulate a DRAM buffer cache when considering the HP trace.

With respect to hardware characteristics, we consider two magnetic disks: a Western Digital Caviar Ultralite CU140, and a Quantum Go•Drive. The CU140 is available with lightweight mobile systems such as the Hewlett-Packard OmniBook 300 PC, and typically spins up the disk in about 1s. However, the disk may take a maximum of 5s to respond after spinning down [17]. Our simulator considers average behavior, so we model the

Additive		Multiplicative	
α_a	β_a	α_m	β_m
2.00	−1.00	1.5	0.5
5.00	−1.00	1.5	0.75
1.00	−0.50	2	0.75
1.00	−0.25	1.25	0.9
2.00	−0.25	1.5	0.9

(a) Adjustment values. The left two columns list the additive values studied in this paper, while the right two list the multiplicative values.

Starting value	Minimum value $T_{\min}$	Maximum value $T_{\max}$
5	2	10
5	2	30
10	5	30
30	10	∞

(b) Ranges of spin-down threshold studied in this paper. Maximal values used for a given configuration depend on the value of δ_M for the disk and are set to $\min(T_{\max}, \delta_M/\rho)$. If the nominal starting value is greater than the maximum it is adjusted to be the mean of the minimum and maximum.

Table 2: Parameters for adaptive spin-down (times in seconds). The cross-product of the sets of parameters was used to drive the simulations; that is, each of the 5 combinations of (α_a, β_a) and 5 combinations of (α_m, β_m) in Table (a) is used with each of the 4 sets of values in Table (b), giving 40 sets of adaptive parameters to drive the simulator.

CU140 by charging 2.5s to spin down the CU140 and 1s to spin it up again. The Go•Drive is a bigger disk and takes 6s to spin down and 2.5s to spin up [15].

We varied the acceptability ratio among three values: 0.02, 0.05, and 0.2. Putting aside for a moment the issue of a request arriving at the disk while it is spinning down, a ratio of 0.05 means that a CU140 would have to be idle for 20s for a spin-up delay of 1s to be acceptable, while for the Go•Drive idle time would normally have to be 50s for the 2.5s delay to be acceptable.[3] Based on subjective personal experience, these idle-time requirements seem like a fair trade-off between delay and energy savings, and we use the ratio of 0.05 as the canonical example of the adaptive approach. A ratio of 0.02 is more restrictive, requiring long idle times to reduce the number of bumps, while a ratio of 0.2 is forgiving: for the CU140, the system need be idle only 5s for a spin-up to be acceptable, though again, accessing the disk while it is spinning down increases the threshold to as much as 17.5s.

Finally, we varied the spin-down policies themselves. The fixed-threshold policies used spin-down thresholds of 2, 5, 10, 30, and 300 seconds. The adaptive ones used additive or multiplicative modifiers within ranges that approximated the fixed-threshold policies. Table 2 shows (a) the modifiers and (b) the starting, minimum, and maximum spin-down thresholds allowed. The values were chosen largely by trial and error; a more formal method of finding an appropriate set of parameters for a given hardware configuration and workload would be useful. Note that the $(\alpha_m = 1.5, \beta_m = 0.5)$ pair contradicts the assumption in Section 3 about the increment upon a bump adjusting the threshold more rapidly than the decrement upon an acceptable spin-up; this issue is considered in Section 5.2 below.

For DRAM and SRAM caching, we used the following approach. Except for the HP trace, which has an implicit DRAM buffer cache, we simulate a 1-Mbyte DRAM buffer cache, which is used to satisfy read requests when possible. One tenth of each trace is simulated before statistics are recorded, in order to prime that cache. All writes go into SRAM; if all blocks in SRAM contain data that are not also stored on disk, a disk write must first take place. We use a cleaning policy similar to the flash memory cleaning policy described in [4], in which the simulator attempts to keep a fraction of the SRAM buffer "clean" at all times, but only if the disk is spinning. In the experiments reported here, SRAM blocks are written to disk either when there is no room for new data, or when the disk is already spinning and fewer than 5% of SRAM is available for new writes; blocks are written out until the next user I/O occurs or 10% of SRAM is free. We consider a more aggressive write policy in Section 5.5.

5 Results

To evaluate different sets of parameters, we plot for each trace the count of bumps against energy consumption. We consider general comparisons of adaptive and fixed-threshold policies in Section 5.1, variations in the adjustment values in Section 5.2, variations in the acceptability ratio in Section 5.3, disk parameters in Section 5.4, and DRAM and SRAM sizes in Section 5.5.

[3] If the disk is accessed just after spinning down, the delay could be as high as 3.5s for the CU140 and 8.5s for the Go•Drive, resulting in minimum idle times of 70s and 170s respectively when $\rho = 0.05$.

5.1 Adaptive versus Fixed Thresholds

WINDOWS

Figure I shows the effect of spin-down policies on energy consumption and bumps, using the Windows trace on the CU140 disk with $\rho = 0.05$. For this trace/hardware configuration, a fixed threshold of 2s consumes the least energy; this follows from previous studies [5, 14]. However, the short threshold results in over 50 bumps over a 1.5 hour period.

Increasing the fixed threshold or moving to an adaptive policy can decrease the number of bumps in exchange for higher energy consumption. At the extreme case, a spin-down threshold of 30s results in no bumps, but also an increase of 48% in energy consumption. Compared to the 2s threshold, a fixed threshold of 10s decreases bumps by two-thirds and increases energy by just 15%.

Figure 1 demonstrates that adaptive policies span a roughly linear range of points between the fixed-threshold policies of 5s and 30s thresholds, depending on the parameters of the adaptive policy. Generally they are in the "desirable region" (below or to the left of comparable fixed-threshold points).

For instance, consider an adaptive policy that varies its threshold between 5 and 30s, increasing the threshold by $\alpha_a = 2$s upon a bump and reducing it by $\beta_a = 1$s upon an acceptable spin-up. (This point is marked by the solid arrow in Figure 1.) This configuration increases energy by just 8% compared to the energy-optimal 2s fixed threshold, and decreases the number of bumps by 65%.

As another example (shown as the dashed arrow), consider an adaptive policy that varies the threshold between 5s and 30s, multiplies the threshold by $\alpha_m = 1.5$ on a bump, and multiplies it by $\beta_m = 0.5$ on an acceptable spin-up. Compared to the 5s fixed threshold policy, this adaptive policy consumes just 0.5% more energy while reducing bumps by a third. Each of the other multiplicative policies within the 5–30s range consumes slightly more energy and encounters slightly fewer bumps, except for the ($\alpha = 1.25, \beta = 0.9$) point, which both consumes more energy and encounters more bumps. This point is shown by the unfilled circle above and to the right of the point shown by the dashed line.

MACINTOSH

Figure 2 shows the Macintosh trace running on the CU140, with $\rho = 0.05$. In this case, except for a couple of points, adaptive spin-down does not provide a demonstrable improvement beyond the ability to interpolate and choose an arbitrary point between different fixed thresholds. With a fixed threshold of 5s, the disk consumes about 6250J and hits 370 bumps. One multiplicative adaptive policy (indicated by the arrow), varying between 2–10s, consumes 6300J (1% more) and reduces bumps to 290 (22% less). But many of the adaptive points cluster near the fixed-threshold curve, consuming about 10% less energy with a small increase in bumps, relative to the curve defined by the simulated fixed-threshold points.

HP

Figure 3 shows the HP trace running on the CU140, with $\rho = 0.05$. At the point where the fixed spin-down threshold is low (2–10s), the adaptive policies follow the fixed curve closely. However, this figure shows that rather than a fixed 10s threshold, one should use an adaptive policy (varying between 10–70s, $\alpha_a = 2$, $\beta_a = -0.2$, indicated by the arrow) that would reduce bumps from around 200 to 100 with only a 3% increase in energy. Moving to a fixed 30s threshold reduces bumps by a similar amount but increases energy more (18%).

5.2 Adjustment Values

Figures 1–3 have shown basic comparisons between adaptive policies, grouped into ranges within which their thresholds are allowed to vary, and fixed-threshold policies. By comparison, Figure 4 graphs bumps against energy consumption for the Windows trace with the adaptive policies grouped by adjustment value rather than range (as was done in Figure 1) . The additive policies are shown with solid marks and the multiplicative ones are shown with unfilled marks.

In this case, the multiplicative policies tended to result in points closer to the upper left region of the graph, i.e., similar to a fixed threshold of 2–10s, while the additive policies tended more toward the center of the graph (comparable to 5–30s). The clear triangles, which show modifiers that multiply the threshold by $\alpha_m = 1.5$ upon a bump and by $\beta_m = 0.5$ upon an acceptable spin-up, support the argument made in Section 3 that the penalty for a bump should outweigh the adjustment when a bump does not occur. When the threshold range is restricted to be low (2–10s or 5–10s), the ($\alpha_m = 1.5, \beta_m = 0.5$) policy increases bumps without a corresponding decrease in energy, compared to both additive and other multiplicative parameters. When the range is closer to the right of the graph, the distinction is not as clear, but compared to the ($\alpha_m = 1.5, \beta_m = 0.9$) policy, the ($\alpha_m = 1.5, \beta_m = 0.5$) policy increases bumps by 30% while reducing energy just 2%.

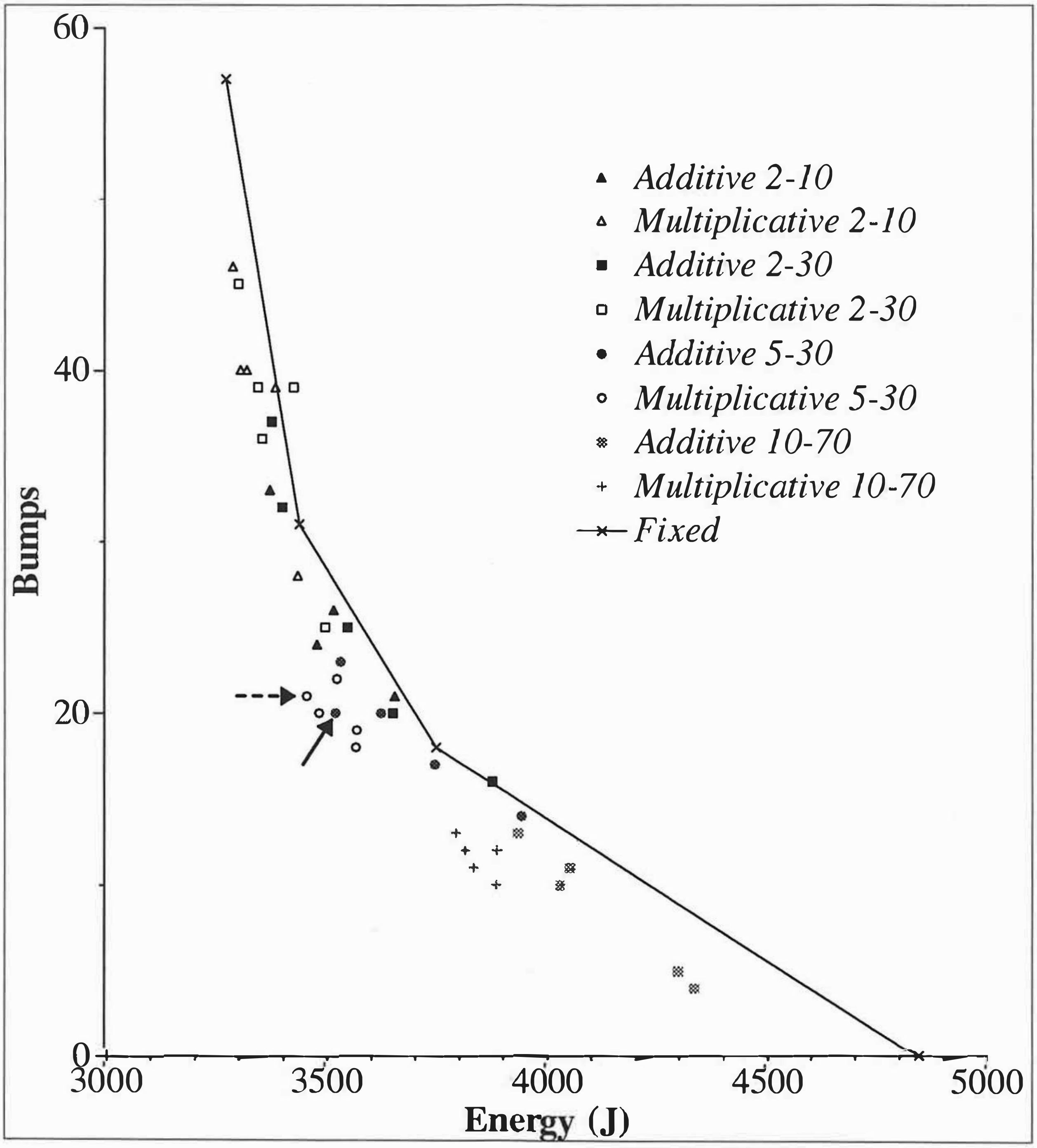


Figure 1: Simulation results comparing bumps and energy consumption, for adaptive and fixed-threshold spin-down policies and an acceptability ratio of 0.05, run on the **Windows** trace and a CU140 disk. In this and subsequent figures, a line connects **x**-marks that represent fixed-threshold policies. Here they show thresholds of 2s, 5s, 10s, and 30s (increasing from left to right). A fixed threshold of 300s, not shown, consumes about 10600J with 0 bumps, compared to 4800J and 0 bumps for the 30s threshold. The adaptive configurations varied in the minimum and maximum spin-down threshold allowed (e.g., 2–30s) and the adjustments to the threshold as shown in Table 2. They are grouped by their allowable ranges, and here the effect of differences in adjustment values within a range is not shown. The arrows point out specific points discussed in the text.

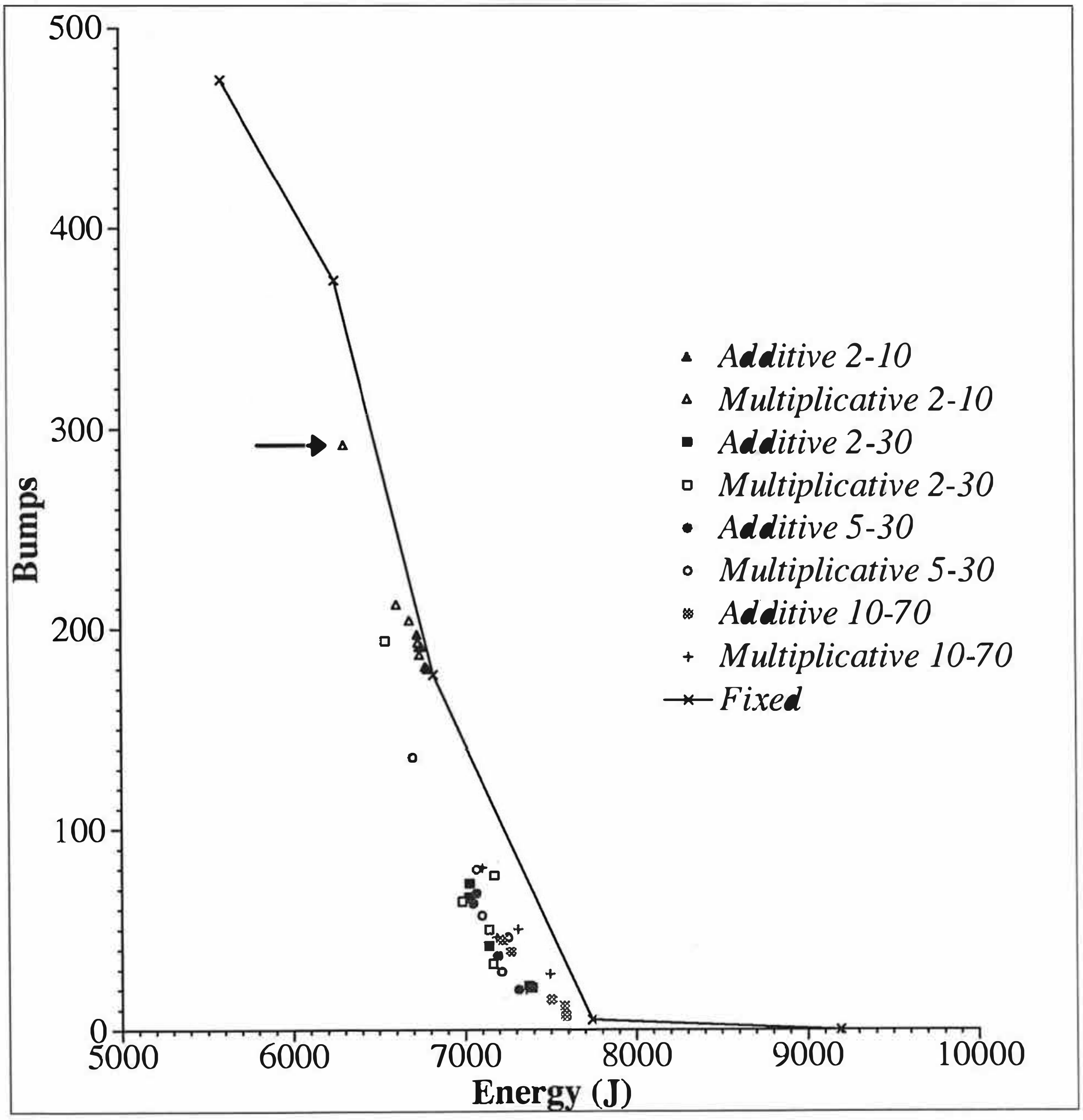


Figure 2: Simulation results comparing bumps and energy consumption, for adaptive and fixed-threshold spin-down policies and an acceptability ratio of 0.05, run on the **Macintosh** trace and a CU140 disk. The line with **x**-marks represents fixed-threshold policies using thresholds of 2s, 5s, 10s, 30s, and 300s (increasing from left to right).

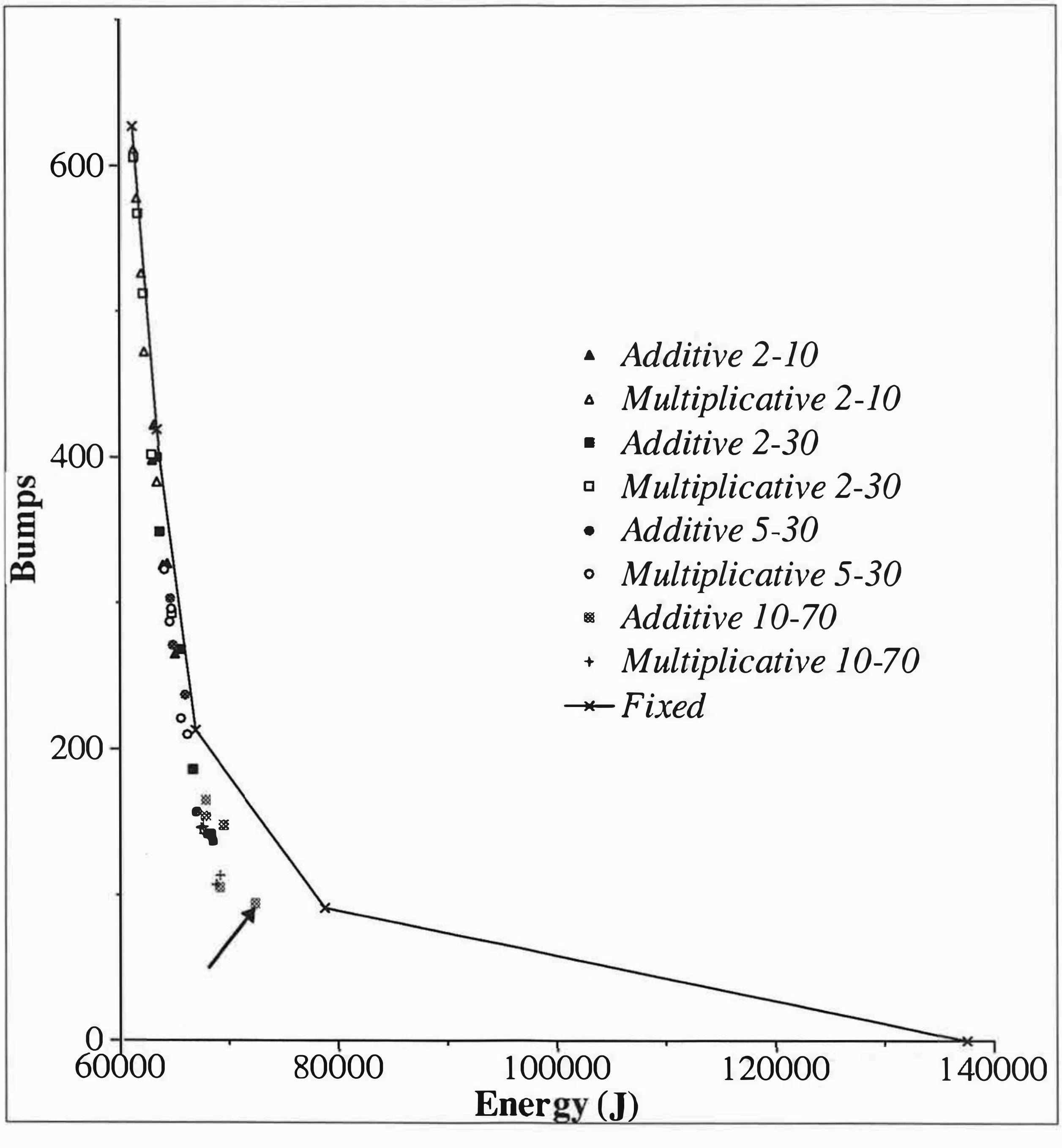


Figure 3: Simulation results comparing bumps and energy consumption, for adaptive and fixed-threshold spin-down policies and an acceptability ratio of 0.05, run on the **HP** trace and a CU140 disk. A line connects x-marks that represent fixed-threshold policies with thresholds of 2s, 5s, 10s, 30s, and 300s (increasing from left to right).

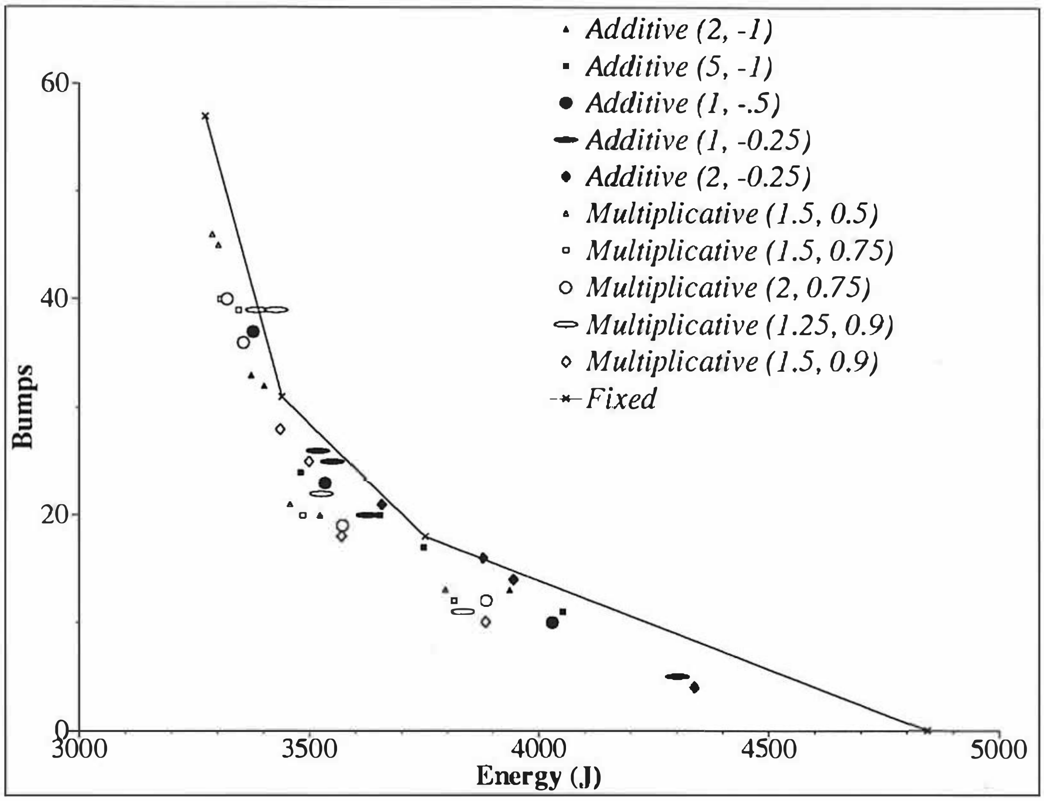


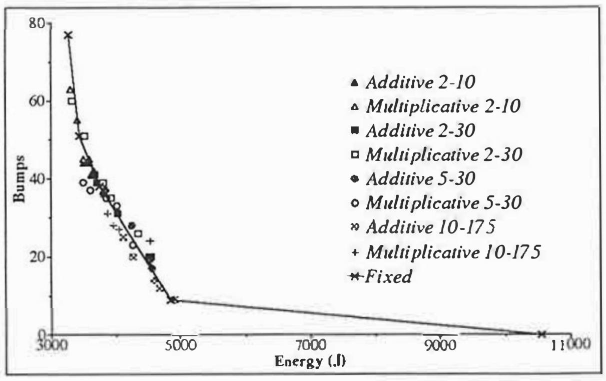
Figure 4: Simulation results comparing bumps and energy consumption, for adaptive and fixed-threshold spin-down policies and an acceptability ratio of 0.05, run on the **Windows** trace and a CU 140 disk. Here the points are grouped by the *adjustment values* rather than the ranges over which the adaptive policies varied. The line connecting x-marks represents fixed-threshold policies with thresholds of 2s, 5s, 10s, and 30s (increasing from left to right); the 300s threshold is omitted. The two values following the policy name specify α and β.

5.3 Varying ρ

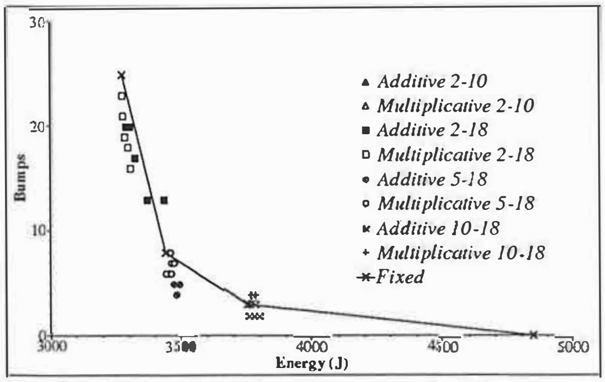
The graphs in Section 5.1 compare adaptive and fixed-threshold policies when the acceptability ratio is 0.05. Figure 5 graphs bumps versus energy consumption for each trace with $\rho = 0.02$ and $\rho = 0.2$. Each pair of graphs for a single trace may be compared with each other and with the graph for $\rho = 0.05$ given previously. (The level of detail in these graphs permits the reader to discern trends among the different traces and policies, but not individual points. Points of interest are discussed in the text, and the raw data for the graphs are available as discussed at the end of the paper.)

For the Windows trace, decreasing ρ results in the adaptive points clustering more closely along the line connecting the fixed-threshold points, but the points farthest to the left (corresponding to fixed thresholds of 2–5s) show adaptive points with 20% reductions in bumps and no increase in energy. Increasing ρ results in the greatest advantage for the adaptive policies: for example, a 2-second fixed threshold results in 3280J and 25 bumps, while one adaptive policy results in 3310J (1% more) and 16 bumps (36% less). A 5-second threshold (3440J, 8 bumps) similarly compares to an adaptive one (3480J, 4 bumps). Of course, the graph shows that there are other adaptive points that are closer to the fixed-threshold results.

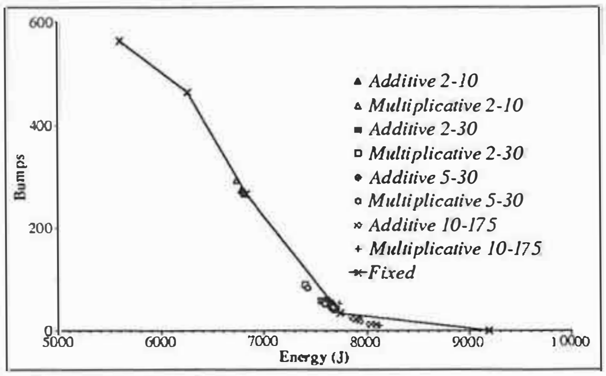
The Macintosh and HP traces show similar effects from changes in the acceptability ratio. Decreasing the ratio, thereby making bumps more common, results in adaptive policies coming fairly close to the fixed-threshold policy. In each case, the jump from 30s to 300s decreases the number of bumps to 0 at a substantial increase in energy consumption, while the adaptive points track the original curve (with fixed thresholds ranging from 2s to 30s). Increasing the acceptability ratio, however, results in adaptive points that represent significant decreases in the number of bumps without significant increases in energy consumption. The Macintosh trace is of particular interest, because the number of bumps barely drops when the fixed threshold moves from 5s to 10s,



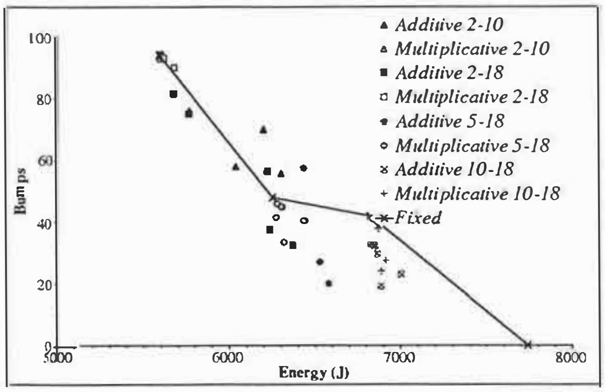
(a) Windows, $\rho = 0.02$



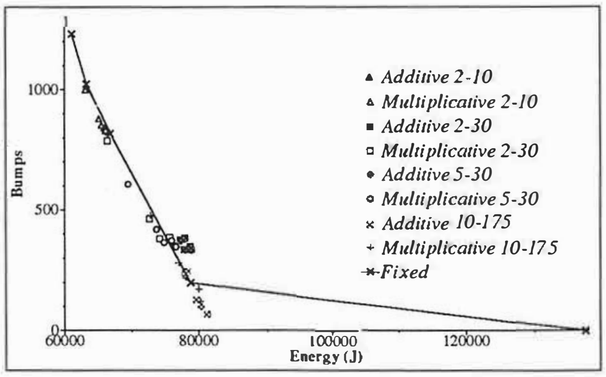
(b) Windows, $\rho = 0.2$



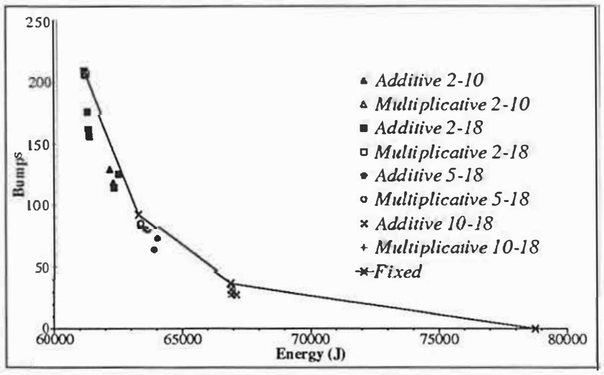
(c) Macintosh, $\rho = 0.02$



(d) Macintosh, $\rho = 0.2$



(e) HP, $\rho = 0.02$



(f) HP, $\rho = 0.2$

Figure 5: Simulation results varying ρ, for each trace. Other parameters are as described in the preceding figures. The scale of each axis varies to show the graphs in the greatest detail. For each trace, when $\rho = 0.2$, there are no bumps with a fixed 30s threshold, and the 300s fixed-threshold run is omitted.

though several adaptive points closely follow the curve one would extrapolate from the 2s and 5s thresholds.

5.4 Varying the Disk

The simulations presented above are based on the Caviar Ultralite CU140, which spins down and up quickly by comparison to some others, such as the Quantum Go•Drive. Figure 6 graphs bumps versus energy consumption for the Macintosh trace with the Go•Drive when $\rho = 0.05$, and it shows an interesting effect of the interaction between disk interarrival times and spin-up costs. For this configuration, the fixed-threshold policy with minimal energy consumption used a 30-second spin-down threshold, rather than 2s. As expected, with a long fixed threshold, the number of bumps is relatively low (34 in this case). Moving to a 10-second threshold increases energy by 4% and bumps by nearly a factor of 8. The increase in energy is a result of the overhead of spinning up the Go•Drive: the break-even point T_d for this disk is 14.9s [5], and spinning it down after 10s of inactivity reduces energy only if it will not be accessed for *another* 15s. Shorter thresholds of 2s and 5s also increased energy consumption by 7% and 9% respectively, compared to the 30s threshold, and increased bumps by factors of 17 and 14 respectively. On the other hand, the adaptive policies generally did about as well as the 30s fixed-threshold policy, or better. The ones that were constrained to spin down within 2–10s increased bumps by as much as 100% without a substantial drop in energy consumption, but the adaptive policies that ranged from 10–175s reduced the number of bumps by up to a third with only a 2–3% increase in energy consumption.

The Macintosh trace is anomalous in that a small (32-Kbyte) write buffer and moderate (1-Mbyte) DRAM cache absorbs enough I/Os to allow the disk to spin down with a short threshold, but not so many that it isn't likely to spin up again quickly relative to T_d. The Windows trace, shown in Figure 7, is more typical, and has a distribution of adaptive points similar to the CU140. For example, one adaptive point represents a 1% increase in energy consumption and a 43% decrease in bumps, compared to the 2s threshold, and a 0.5% increase in energy and 14% decrease in bumps compared to the 5s threshold.

5.5 Caching Effects

As mentioned in Section 2.1, both DRAM caching and SRAM buffering can reduce energy consumption and improve performance.

Figure 8 graphs bumps versus energy consumption for the Macintosh trace, similar to Figure 2 in all ways except DRAM size, which is increased here to 2 Mbytes. Adding DRAM beyond the first Mbyte does not appreciably affect either the fixed-threshold or adaptive policies.

Figure 9 shows the effect of SRAM size on the Macintosh trace. Comparing Figure 9(a), with no SRAM buffer at all, to Figure 2, with a 32-Kbyte SRAM buffer, demonstrates that for this trace a small SRAM buffer dramatically reduces energy,[4] but at the cost of a large number of bumps to get the best energy savings. Both graphs show that adaptive policies improve moderately over fixed-threshold policies, without SRAM size being a great factor. Figure 9(b) indicates that a large SRAM buffer further reduces both energy consumption and bumps, with adaptive policies continuing to show improvements over fixed ones.

We have mentioned that writes to SRAM are deferred when possible. Figure 10 shows the effect on the HP trace of writing SRAM blocks to disk more "aggressively," ignoring the need to spin up the disk, and writing all data through to disk quickly (not just 10% of the SRAM buffer). Here, all modified data goes through to disk at the earliest possible moment, as long as there are not other user I/Os taking place. This figure, when compared to Figure 3, demonstrates that deferring disk writes reduces power consumption but generally results in more bumps when the spin-down threshold is small. This is not surprising: since the writes are not clustered as much, more of them follow a brief period of inactivity. For this particular trace, which has periodic disk writes resulting from the UNIX 30-second *sync* policy, a 30s fixed-threshold policy behaves especially poorly, while the adaptive policies do not suffer from this anomalous behavior.

Changing the SRAM write-back policy for the Windows and Macintosh traces does not generate any pathological thresholds; in fact, for the Windows trace, which is only 20% writes, it has a minimal effect. Figure 11 shows the same experiment as the Macintosh run in Figure 2, but without the deferred writes. For the fixed 2s threshold, energy consumption increases by 67% without deferred writes, but bumps decrease by 80%. But to get down to nearly no bumps requires over 10,000J, compared to under 8000J in the deferred-write case.

5.6 Discussion

The preceding analyses demonstrate that the effectiveness of adaptive policies depends on many factors. Generally speaking, if the user's goal is to minimize energy while paying at least some attention to spin-up delays, the adaptive policies that permit the threshold to go as

[4] This contrasts with the result reported in [4], which did not buffer writes to a spun-down disk as effectively as possible, and showed only about a 20% improvement due to SRAM buffering.

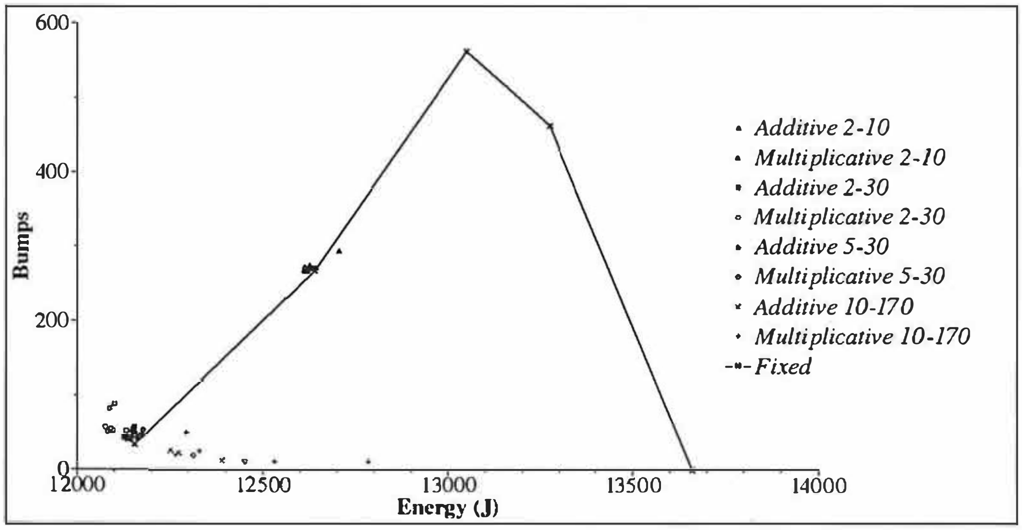


Figure 6: Simulation results comparing bumps and energy consumption, for adaptive and fixed-threshold spin-down policies and an acceptability ratio of 0.05, run on the **Macintosh** trace and a **Quantum Go•Drive** disk. A line connects x-marks that represent fixed-threshold policies, but with anomalous behavior: this time the thresholds are 30, 10s, 2s, 5s, and 300s respectively, increasing from left to right.

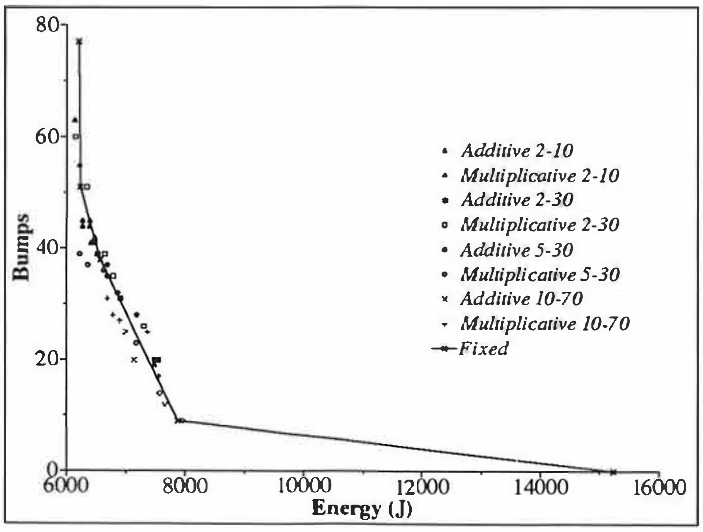


Figure 7: Simulation results comparing bumps and energy consumption, for adaptive and fixed-threshold spin-down policies and an acceptability ratio of 0.05, run on the **Windows** trace and a **Quantum Go•Drive** disk. The line with x-marks represents fixed-threshold policies using thresholds of 2s, 5s, 10s, 30s, and 300s (increasing from left to right).

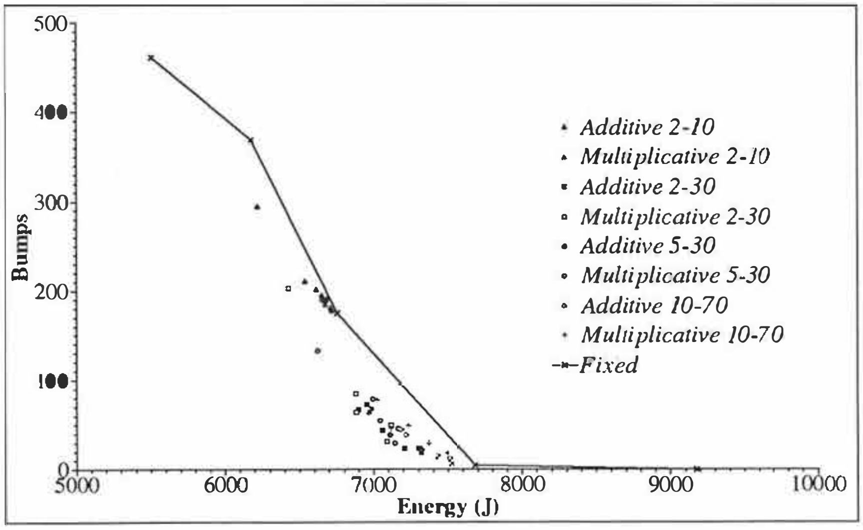
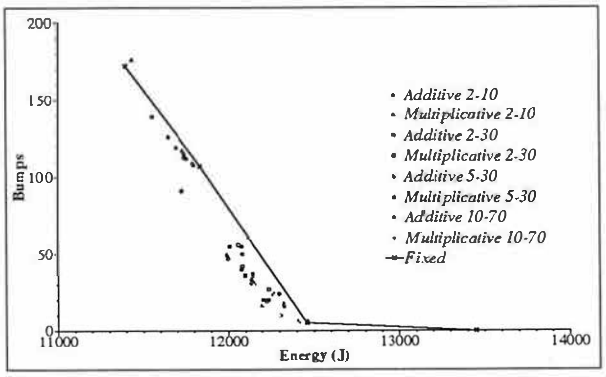
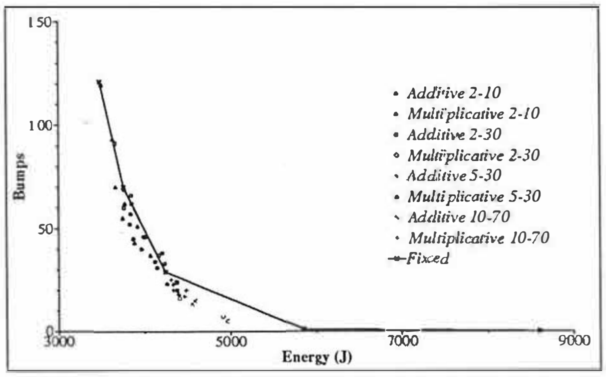


Figure 8: Simulation results comparing bumps and energy consumption, for adaptive and fixed-threshold spin-down policies and an acceptability ratio of 0.05, run on the **Macintosh** trace, a CU140 disk, and **2 Mbytes** of DRAM. The line with **x**-marks represents fixed-threshold policies using thresholds of 2s, 5s, 10s, 30s, and 300s (increasing from left to right).



(a) No SRAM. Here, the point with a fixed 2-second spin-down is omitted, as it consumes more energy than a 5s or 10s threshold and results in over 600 bumps.



(b) 1 Mbyte of SRAM.

Figure 9: Simulation results varying SRAM size for the **Macintosh** trace running on the CU140 with $\rho = 0.05$. The line with **x**-marks represents fixed-threshold policies using thresholds of (2s), 5s, 10s, 30s, and 300s (increasing from left to right).

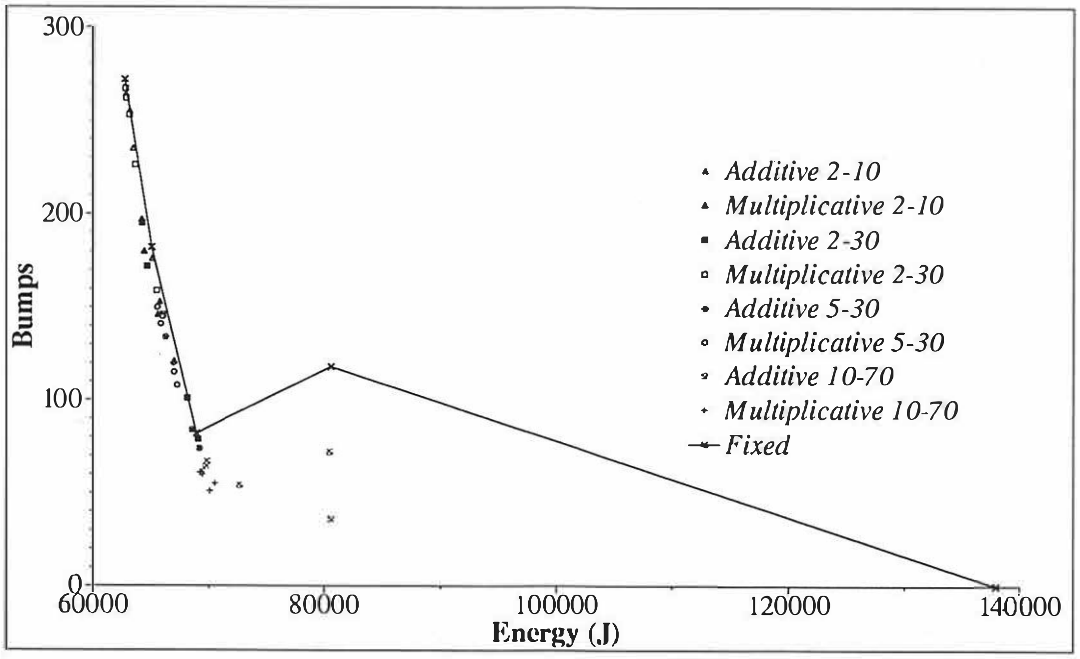


Figure 10: Simulation results for the **HP** trace running on the CU 140 with $\rho = 0.05$ and writes to SRAM going through immediately to disk. The line with x-marks represents fixed-threshold policies using thresholds of 2s, 5s, 10s, 30s, and 300s (increasing from left to right). Spinning down the disk after 30s of inactivity results in greater energy consumption and more bumps than a 10s threshold, due to periodic writes.

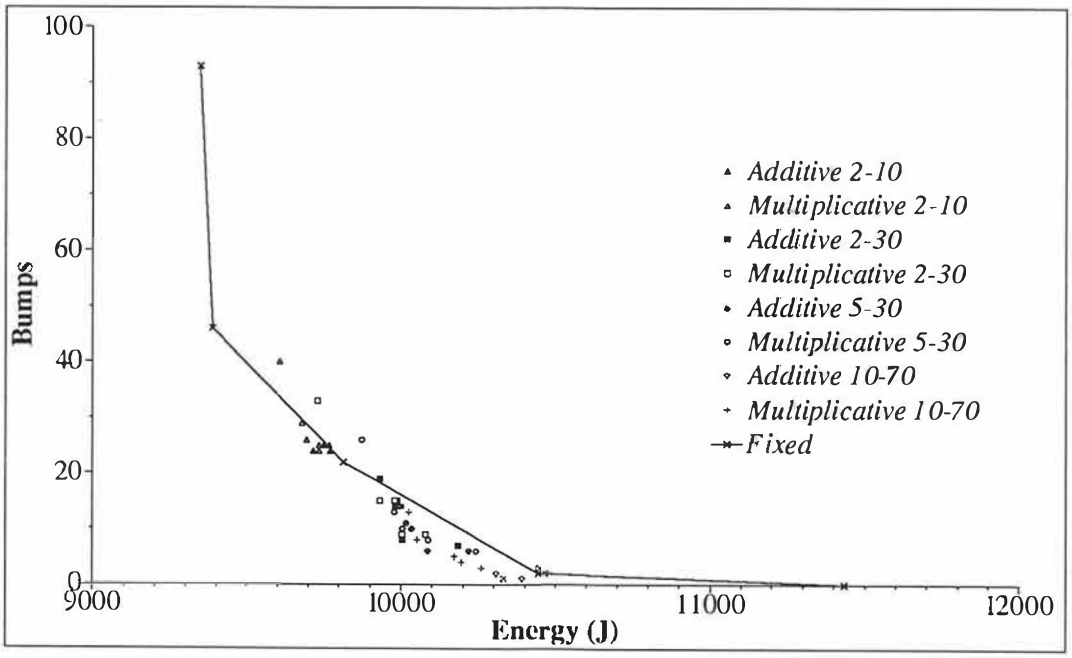


Figure 11: Simulation results for the **Macintosh** trace running on the CU140 with $\rho = 0.05$ and writes to SRAM going through immediately to disk. The line with x-marks represents fixed-threshold policies using thresholds of 2s, 5s, 10s, 30s, and 300s (increasing from left to right).

low as 2s are useful. They encounter fewer bumps than a fixed-threshold policy of 2–5s but for only a little more energy. The more willing the user is to tolerate spin-up delays, the more effective adaptive policies can be.

We mentioned in the introduction that it is hard to improve both bumps and energy consumption using these techniques. To reduce energy while simultaneously reducing the number of bumps compared to a fixed threshold policy, we would need to spin down the disk almost immediately whenever a spin-up would be acceptable. Intuitively, this requires a sophisticated prediction of acceptable spin-ups; prediction of interarrival times at disk for use in disk spin-down has proven to be difficult [5], though it warrants further study.

Instead, we have described in this paper a technique to vary the threshold dynamically to gain on one of these metrics, bumps, by compromising on the other, energy consumption. Users of mobile computers can already make this tradeoff simply by varying the fixed spin-down threshold, but changes in workload over time can result in changes to the proper place to make this tradeoff. By varying the threshold based on recent history instead of a static parameter, adaptive policies can react to these changes. At times, an adaptive policy may avoid cases when the disk spins for a while, and then spins down just before an access. If these cases can be avoided, the adaptive policy will save both energy and bumps. In other cases, when comparing an adaptive policy to a fixed-threshold one, there will be times when by spinning down the disk earlier some energy is saved in exchange for a bump that the fixed-threshold policy avoided, or vice-versa. One metric or the other will increase relative to the fixed-threshold policy.

6 Related Work

Most of the prior work in this area has focussed on what threshold to use for the fixed-threshold policy [5, 7, 14]. Generally speaking, a spin-down threshold of 2–5s consumed the least energy of all fixed thresholds, but resulted in several spin-up delays per hour.

Wilkes hypothesized that it would be effective to use a weighted average of a few previous interarrival times to decide when to spin down the disk on a mobile computer. He noted as well that if inactive intervals were of roughly fixed duration, the disk could be spun up in advance of the expected time of the next operation [18]. If access patterns are not so consistent, however, these techniques may not prove to be helpful.

Golding, et al., studied idle-time detection and prediction in a more general framework [6]. They considered a number of prediction methods, including arithmetic and geometric adjustments of a predicted interval. Although they note the applicability of their taxonomy to powering down components on portable computers, they reported the effect of different methods only in the context of the TickerTAIP simulation system [16]. Also, they separate the prediction of *when* an idle period will arrive from the prediction of its *duration*. Here, our spin-down threshold is a prediction of how long the system should wait within an idle period before deciding that the remaining duration of the idle period is long enough to justify spinning down the disk.

On-line optimization of when to spin down the disk is similar to detecting how long to hold a virtual circuit open [12] or whether to spin on a lock or context switch [10], and can be modeled by a sequence of rent-to-buy decisions [13]. Our adaptive spin-down policies are similar to the "random walk" method described by Karlin, et al., which they reported performed almost as well as the optimal on-line policy and was more efficient than other adaptive policies they studied. A "profiling" approach (modeling the distribution) works better than a "random walk" approach for the virtual circuit problem [11], and we plan to explore profiling in the context of this problem.

7 Summary and Future Work

Adaptive disk spin-down allows the user of a mobile computer to trade off the energy consumption gained by spinning down a hard disk against the inconvenience of spinning the disk up again. It uses recent history to adjust the threshold for spinning the disk down, based on the user's specification. This specification is made in terms of the amount of time the disk must have been idle, compared to the amount of time the user is delayed by a spin-up.

Simulations with three sets of trace data and two magnetic disks indicate that adaptive spin-down offers advantages over a fixed threshold. In some configurations, adaptive policies eliminate up to a third of all undesirable spin-ups, or more, with only a marginal increase in energy consumption, such as 3%. Others represent intermediate points that could be obtained by varying a fixed-threshold policy, and still others represent undesirable points that are worse than fixed-threshold policies.

There are several directions that may be explored further. No single set of parameters (increments or ranges) is uniformly the best across each configuration, or uniformly undesirable; obviously, the differences in adjustment parameters (additive versus multiplicative, values and criteria for adjustment, and threshold bounds) should be further considered. The taxonomy described by Golding, et al. [6] would be helpful in this regard. The definition of a "bump" may warrant further elaboration, for example to take the energy savings from spinning down the disk into account (so a disk that has just been spun down and then spins up again would constitute a bump regardless of when the user last accessed it). Finally, these policies should be implemented in real systems and tested empirically.

Acknowledgments

W. Sproule of MITL gathered the Macintosh traces used in this study. R. Caceres, M. F. Kaashoek, K. Li, and B. Marsh helped with our understanding of the policies and algorithms. S. Keshav helped formulate optimization strategies. B. Krishnamurthy, D. Milojičić, D. Presotto, A. Skarra, J. Trotter, and the anonymous referees provided comments on earlier drafts.

Availability

The raw data for the graphs in this paper are available via anonymous *ftp* from *research.att.com* in the directory *dist/douglis/mlic95*.

References

[1] Mary Baker, Satoshi Asami, Etienne Deprit, John Ousterhout, and Margo Seltzer. Non-volatile memory for fast, reliable file systems. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 10–22, Boston, MA, October 1992. ACM.

[2] Connectix Corporation, San Mateo, CA. *CPU Connectix PowerBook Utilities Version 2.0 Addendum*, April 1993.

[3] Dell Computer Corporation. *Dell System 320SLi User's Guide*, June 1992.

[4] Fred Douglis, Ramón Cáceres, Brian Marsh, Frans Kaashoek, Kai Li, and Joshua Tauber. Storage alternatives for mobile computers. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 25–37. USENIX Association, November 1994.

[5] Fred Douglis, P. Krishnan, and Brian Marsh. Thwarting the Power Hungry Disk. In *Proceedings of 1994 Winter USENIX Conference*, pages 293–306, San Francisco, CA, January 1994.

[6] Richard Golding, Peter Bosch, Carl Staelin. Tim Sullivan, and John Wilkes. Idleness is not sloth. In *Proceedings of the USENIX 1995 Technical Conference on UNIX and Advanced Computing Systems*, pages 201–212, New Orleans, LA, January 1995.

[7] Paul Greenawalt. Modeling Power Management for Hard Disks. In *Proceedings of the Symposium on Modeling and Simulation of Computer and Telecommunication Systems*, 1994.

[8] Hewlett-Packard. Kittyhawk power management modes. Internal document, April 1993.

[9] B. Johnson. *EPA Energy Start Computers Program*. Environmental Protection Agency, 1993. Basic Information Package.

[10] Anna R. Karlin, Kai Li, Mark S. Manasse, and Susan Owicki. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 41–55. Association for Computing Machinery SIGOPS, October 1991.

[11] S. Keshav. Personal Communication, 1994.

[12] S. Keshav, C. Lund, S. J. Phillips, N. Reingold, and H. Suran. An empirical evaluation of virtual circuit holding time policies in IP-over-ATM networks. *IEEE Journal on Selected Areas in Communications*, 1995. To appear.

[13] P. Krishnan, P. Long, and J. S. Vitter. On-line rent-to-buy in probabilistic environments. Technical Report CS–1995–08, Duke University, 1995.

[14] Kester Li, Roger Kumpf, Paul Horton, and Thomas Anderson. A Quantitative Analysis of Disk Drive Power Management in Portable Computers. In *Proceedings of the 1994 Winter USENIX*, pages 279–291, San Francisco, CA, 1994.

[15] Quantum. *Go•Drive 60/120S Product Manual*, May 1992.

[16] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, March 1994.

[17] Western Digital. *Caviar UltraLite CU140 Technical Reference Manual*, May 1993.

[18] John Wilkes. Predictive power conservation. Technical Report HPL-CSP-92-5, Hewlett-Packard Laboratories, February 1992.

[19] Zenith Data Systems, Groupe Bull. *MastersPort 386SL/386SLe Owners Manual*, 1991.

ISBN 1-880446-69-3